O'REILLY®

# 基于Ionic的移动App开发

Mobile App Development with Ionic 2

Chris Griffith 著
杨宏焱 译

中国电力出版社

# 基于Ionic的移动App开发

Chris Griffith 著

杨宏焱 译

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

O'Reilly Media, Inc. 授权中国电力出版社出版

中国电力出版社

**图书在版编目（CIP）数据**

基于Ionic的移动App开发 /（美）克里斯·格里菲斯（Chris Griffith）著；杨宏焱译. — 北京：中国电力出版社，2017.12

书名原文：Mobile App Development with Ionic 2

ISBN 978-7-5198-1424-3

I. ①基… II. ①克… ②杨… III. ①移动终端－应用程序－程序设计 IV. ①TN929.53

中国版本图书馆CIP数据核字(2017)第291696号

北京市版权局著作权合同登记 图字：01-2017-5675号

---

出版发行：中国电力出版社

地　　址：北京市东城区北京站西街19号（邮政编码100005）

网　　址：http://www.cepp.sgcc.com.cn

责任编辑：刘 炽（liuchi1030@163.com）

责任校对：朱丽芳

装帧设计：Karen Montgomery，张 健

责任印制：蔺义舟

---

印　刷：三河市百盛印装有限公司

版　次：2017年12月第一版

印　次：2017年12月北京第一次印刷

开　本：750毫米×980毫米　16开本

印　张：18.5

字　数：351千字

印　数：0001—3000册

定　价：68.00元

---

# O'Reilly Media, Inc.介绍

O'Reilly Media通过图书、杂志、在线服务、调查研究和会议等方式传播创新知识。自1978年开始，O'Reilly一直都是前沿发展的见证者和推动者。超级极客们正在开创着未来，而我们关注真正重要的技术趋势——通过放大那些“细微的信号”来刺激社会对新科技的应用。作为技术社区中活跃的参与者，O'Reilly的发展充满了对创新的倡导、创造和发扬光大。

O'Reilly为软件开发人员带来革命性的“动物书”；创建第一个商业网站（GNN）；组织了影响深远的开放源代码峰会，以至于开源软件运动以此命名；创立了Make杂志，从而成为DIY革命的主要先锋；公司一如既往地通过多种形式缔结信息与人的纽带。O'Reilly的会议和峰会集聚了众多超级极客和高瞻远瞩的商业领袖，共同描绘出开创新产业的革命性思想。作为技术人士获取信息的选择，O'Reilly现在还将先锋专家的知识传递给普通的计算机用户。无论是通过书籍出版，在线服务或者面授课程，每一项O'Reilly的产品都反映了公司不可动摇的理念——信息是激发创新的力量。

## 业界评论

“O'Reilly Radar博客有口皆碑。”

——Wired

“O'Reilly凭借一系列（真希望当初我也想到了）非凡想法建立了数百万美元的业务。”

——Business 2.0

“O'Reilly Conference是聚集关键思想领袖的绝对典范。”

——CRN

“一本O'Reilly的书就代表一个有用、有前途、需要学习的主题。”

——Irish Times

“Tim是位特立独行的商人，他不光放眼于最长远、最广阔的视野并且切实地按照Yogi Berra的建议去做了：‘如果你在路上遇到岔路口，走小路（岔路）。’回顾过去Tim似乎每一次都选择了小路，而且有几次都是一闪即逝的机会，尽管大路也不错。”

——Linux Journal

# 目录

# 序

在2013年，我们小组正忙于几个拖曳式工具的开发，用在两个最流行的移动和桌面Web框架上：jQuery Mobile和Bootstrap。我们发现在Web开发中的可重用组件和框架发生了快速增长，正在创建更好和更具包容性的工具，让它们的使用变得更加简单。

刚好在那个时候先后发布了iPhone 5和iOS 7，Web性能得到了明显提升，新的Web API也开放了，移动浏览器App的性能和特性达到了前所未有的程度。我们开始设想，能不能利用这些新特性搭建出一个框架，提供类似原生一样的UI工具包，让Web开发者能够用标准的浏览器技术创建出原生品质的App？你可能会说，这是“移动版的Bootstrap”吧？

恰逢此时，Angular 1被广泛使用于各种Web开发环境，完美地解决了JavaScript和HTML Web组件的重用问题。我们决定搭建一个移动优先的Web UI框架，使用高速增长的Angular 1框架，来实现交互性和可分发。

Ionic的第一版发布于2013年末，它吸引了Web开发者的兴趣，这个项目使其在GitHub上的star数目和用npm安装的数目快速增长。一年半后，创业公司、开发团队和企业用户用这个项目开发超过100万个App。

2015年，JavaScript在一夜之间发生了翻天覆地的变化。ES 5，这个从Web 2.0时代就广为人知的JavaScript版本变成了过去式。取而代之的是ES 6，新一代的JavaScript版本产生了，具有全新的面向对象开发特性，如新的sharing和

loading 模块，更简洁的语法等。整个 JavaScript 世界因为浏览器运行时沸腾不已，开发者争先恐后地用上了 ES 6。

转译器被编写出来了，用于将新的 JavaScript 语法转译成 ES 5 语法以便浏览器能够理解。开发者正在摸索将 JavaScript 库分发为可重用模块的最佳做法。新的用于编译和发布独立 JavaScript 模块的构建工具不断被发明、抛弃和重建。试图减少错误并使现代 JavaScript 语法更加标准化的新项目出现了，比如 TypeScript 和 Flow。随着 ES 7 以后的实验特性不断被添加到转译器中，在新特性进入生产代码库之前，那些被抛弃和从标准中删除的特性会让保守的 JavaScript 开发者更加懊悔。总之，这是一场混乱。

来自 ES 6 之前时代的框架作者，突然会面临着一个令人生怯的任务，将自定义的抽象换成标准的 ES 6 以后有效的抽象。这些框架，很少有动力开发出像 Angular 1 一样的抽象。对于 Angular 来说，问题十分简单：如何将所有的和框架相关的东西比如 scope、controller、directive 等转换成标准的 JavaScript 类、Web 组件一类的东西呢？

通过 JavaScript 的这次罕见的升级，Angular 团队从第一个主要的 JavaScript 框架的构建中取得了一些经验，并在未来的兼容 Web 和移动开发的框架中应用这些经验。这并不妨碍将 Angular 1 的主要概念映射成非常成熟的 ES 6 概念。事实上，从许多方面来说，它们在 ES 6 中变得更自然。

当得知 Angular 2 要开发出来的时候，我们立即意识到这是一个机会，将从 Ionic 1 的构建以及那 100 万 App 的制作中得到的经验用到新的框架中来。

Ionic 团队从 2015 年春开始编写 Ionic 2。经过一年半的开发，犯了一些错，找到了新的解决方案，进行了大胆的尝试，我们最终兴奋地推出了一个大版本，可用于生产的 Ionic 2。

从表面上看，Ionic 2 和 Ionic 1 一样。组件使用自定义 HTML 标签编写，这些标签被 Ionic 转换成强大的移动组件。Action 被绑定到类的回调上，这个类扮演着 App 中某个页面的控制器角色。项目的编译和测试使用的是相同的命令行工具。外观和主题是类似的，Ionic 2 的外观和 Ionic 1 别无二致。

不同的地方在于底层。Ionic 2 底层是用 TypeScript 和 Angular 2 重写的。所有的 Ionic 代码都是类型化的，在我们自己的代码中明显降低了 bug 和类型问题的出现几率。还有一些让人惊喜的新特性，比如，对于使用 Atom 和 Visual Studio Code 的开发者而言，可以使用行内文档并能轻松重构了。代码更加面向对象了，特别是对于一个 UI 框架来说，尤其有用。结构不再像 Angular 1 那么原始。

Angular 2 为实现在手机上平滑运行进行了重构，采用了一系列手段，比如降低性能消耗，使核心操作（比如变化感知）变得流畅。因此，Ionic 2 App 运行起来比 Ionic 1 App 更快，能够处理更复杂的任务。

Ionic 的目的一直是使用最轻松的方法构建优秀的移动应用。如果我们不能确信在提升功能的同时使 Ionic 变得更易用，我们是不会花费时间和成本重写框架的。我们相信 TypeScript 能够使 Ionic 代码书写和理解起来都更加轻松。我们相信 Angular 2 比 Angular 1 更加容易使用，它需要的域特定语言和知识要少得多。我们相信 Ionic 2 项目更简洁，组织性更好，组件的使用更加简单。

除了底层技术，Ionic 2 还有几个主要的新特性。现在，Ionic 2 通过对材料设计的支持和简化的主题样式，让 App 的外观和运行时的平台相适配。我们的导航系统能够创建出各种灵活的、与原生 App 类似的导航，而这是在浏览器中无法做到的。我们添加了太多的特性、组件和数不清的原生 API。

另外，移动领域在 2016 年还发生了一个剧烈变化。突然之间，移动 Web 又重新流行起来，同时渐进式 Web App 以一种强势的方式进入前台。Google 打通了一个全新的世界，App 运行在浏览器中，不需要安装，不管网络带宽和网络是否连接，都能够提供良好的用户体验，移动开发者对于将移动 Web 作为他们的移动解决方案深信不疑。

使用 Ionic 2 的开发者现在可以在不修改任何代码的情况下迁移到移动 Web。Ionic App 既是一个运行在 iOS 和 Android 设备上的原生 App，也是一个运行在移动 Web 上的渐进式 Web App。一次编写，随处运行。

我们将全部重心放在 Ionic 2，很高兴能够将 Ionic 2 推荐给可用于生产的移动 App。我们希望你能够和我们一样发现它是一个高性能的、灵活的框架，用它编写移动 App 和移动网站比你想象的还要容易。有接近三百万的 App 是用 Ionic

构建的。我们对如何搭建高质量的App框架略知一二，汲取了每一个经验教训，并将它们全部用到了Ionic 2中。

如果你喜欢Ionic 2，希望你也了解一下我们编写的配套工具，它们能给Ionic开发者提供一些便利，包括快速测试工具Ionic View、快速原型创建工具、快速App开发工具Ionic Creator，以及在Ionic Cloud中的一整套轻量级后端集成服务。Ionic正在变成一个和移动开发相关的“一站式”商店。

我代表全体Ionic团队成员，预祝你Ionic 2用得开心，让我们在论坛（*https://forum.ionicframework.com*）中再见！

——Max Lynch
创始人/CEO，Ionic

# 前言

我第一次涉足移动开发领域的事情要从 2007 年说起。当时，我正在应聘 Qualcomm 用户体验团队中的 UX 工程师职位，时逢史蒂夫·乔布斯发布他的第一台 iPhone。我的面试过程几乎变成了一场苹果发布会的讨论。那一天所发生的一切影响了我的整个职业生涯。接下来的十年间，我接触过各种移动开发解决方案。一直以来，我的最终目标是找到一种能够快速开发、让团队能够进行快速测试和快速实践新点子及想法的解决方案。

我曾经做过许多移动原型设计，它们的用户体验都是高度定制化的。需要模仿真实设备原生组件的需求几乎没有。偶尔需要用到原生组件的时候，我都是在方案中用到时重新设计。到最后，原型设计中定制化的成分越来越少，大部分都采用默认设计。我需要找出一种能够提供大量组件的解决方法，因为我不再想自己开发和维护某个定制组件了。

我开始尝试在一些项目中使用 Ratchet 和 TopCoat 这样的库。我在加利福尼亚大学的“圣地亚哥推广计划”中给学生们上一门课叫做“移动开发入门”，是基于 jQuery Mobile（和 PhoneGap Build）的。但是，这些解决方案都不能在创建原型时为我提供丰富的组件库。

我不记得什么时候知道了 Ionic 框架，但我知道它是基于 Apache Cordova 和 AngularJS 的。我曾经录制过两个关于 PhoneGap Build 和 Apache Cordava 的视频课程，但对 AngularJS 了解得非常少。由于原型开发的原因，我通常会对那些大框架望而生怯。不久以后，我看到别的组件库也使用了相同的技术栈。因此，

我决心开始学习 AngularJS 和 Ionic 框架。很快，我就被这两个框架所展示出来的威力所折服，开始在我的解决方案中使用它们。

我开始疯狂学习 Ionic，直到我发布了第一个用 Ionic 编写的商业移动应用 Hiking Guide:Sedona。随后，在 2015 年 10 月，Ionic 2 发布了。这次发布不仅仅是一次升级，同时也是一次重大突破。因此同样的过程开始了：学习新的 Angular 语法，使用 ES 6、TypeScript，甚至我将原来的编辑器升级到了微软的编辑器！在这个过程中 Ionic 2 也在不断成长和趋于成熟。

编写本书是一个漫长和有趣的过程。Ionic 的每个版本的发布都会迫使我认真阅读它的修改日志，评估它对已写和未完成章节的影响。这些都会让我对这个框架有更深的理解。我希望本书成为一本学习使用 Ionic 编写混合移动应用的指南。

Chris Griffth，圣地亚哥，2017 年 1 月

## 本书面向的读者

本书适合准备学习 Ionic 框架的初学者。本书需要你熟悉 JavaScript、HTML 和 CSS。本书会涉及一些 TypeScript、ES 6、Angular 2 和 Apache Cordova 的主要概念，这些内容你可以参考更多相关资源。本书是以 step by step 方式编写的，请放心阅读并学习如何用 Ionic、Angular 和 Cordova 编写混合移动 App 吧。

## 本书结构

本书会带你依次了解 Ionic 框架的每一部分。各章内容简要介绍如下：

- 第 1 章，混合移动 App，介绍混合移动应用的概念。
- 第 2 章，配置我们的开发环境，介绍编写 Ionic 应用程序需要些什么。
- 第 3 章，理解 Ionic 命令行界面，深入介绍 CLI 的功能。
- 第 4 章，Angular 和 TypeScript 基础，介绍 Angular 和 TypeScript 基础知识。
- 第 5 章，Apache Cordova 基础，介绍 Apache Cordova 以及为什么它会被纳入 Ionic 框架的一个组成部分。

- 第 6 章，理解 Ionic，简单介绍一个 Ionic 页面由什么构成。
- 第 7 章，编写我们的 Ionic2Do App，创建一个基于 Firebase 的 to-do 应用程序。
- 第 8 章，创建一个基于 Tab 的 App，使用 Tab 模板编写一个国家公园浏览程序，集成 Google 地图。
- 第 9 章，创建一个天气应用，使用 Forecast.io 天气 API 和谷歌地理编码 API，编写一个带侧边栏菜单的应用程序。
- 第 10 章，调试并测试你的 Ionic 应用，介绍通过一些常用工具解决开发中出现的问题。
- 第 11 章，部署你的应用程序，一步一步教你如何将应用提交到应用商店。
- 第 12 章，探索 Ionic Cloud，探讨由 Ionic 平台提供的其他服务。
- 第 13 章，渐进式 Web 应用，讨论如何用 Ionic 作为一个基础的渐进式 Web App。
- 第 14 章，终章，介绍其他 Ionic 组件并扼要介绍了一些其他资源。
- 附录 A，从 Ionic 1 升级到 Ionic 2，介绍二者间的重要改变。
- 附录 B，理解 Config.xml 文件，介绍和应用程序编译过程有关的各种属性配置。
- 附录 C，Ionic 组件库，罗列每个可用的 Ionic 组件及一般用法。

所有的代码放在 GitHub 上，如果你不想手敲书中的示例代码，或者你想看最新的示例代码，请访问代码库并下载它的源代码。

如果你做过 Ionic 1 的开发，可以跳过第 1~3 章。如果你熟悉 TypeScript 和 Angular 2，请略过第 4 章。如果你会使用 Apache Cordova 或 PhoneGap，请略过第 5 章。

## 在线资源

下列资源对于每个 Ionic 开发者来说都是一个很好的开始，它们总是随手可及的：

- 官方 Ionic API 文档（*http://ionicframework.com/docs/*）。
- 官方 Angular 2 文档（*https://angular.io/docs/t/latest/*）。
- 官方 Apache Cordova 文档（*https://cordova.apache.org/docs/en/latest/*）。
- Ionic 全球 Slack 频道（*http://ionicworldwide.herokuapp.com/*）。

## 本书排版约定

本书的排版方式约定如下：

*斜体（Italic）*

表示新出现的术语、URL、email、文件名及扩展名。

等宽字体 （`Constant Width`）

在代码清单中使用，或者在段落中用于表示程序中的对象，例如变量名、函数名、数据库、数据类型、环境变量、语句和关键字。

粗体等宽字体（**`Constant Width bold`**）

表示命令或需要用户输入的其他文本。

斜体等宽字体（*`Constant Width Italic`*）

表示文本应该由用户自己提供的内容替换，或者内容应根据上下文改变。

表示提示或建议。

表示一般的注意事项。

表示警告或警示。

## 使用示例代码

如果在代码行的尾部看到 ↵，说明这行代码之后是另一行代码。

## O'Reilly Safari

Safari (Safari 图书在线 ) 是一个针对企业、政府、教育机构和个人的会员制培训和参考平台。

成为会员将可以从数据库中查找和浏览数以千计的图书、培训视频、学习路径、交互式教程和组织好的播放列表，这些资料的来源遍及 250 个出版社，如 O'Reilly Media、Harvard Business Review、Prentice Hall Professional、Addison-Wesley Professional、Microsoft Press、Sams、Que、Peachpit Press、Adobe、Focal Press、Cisco Press、John Wiley & Sons、Syngress、Morgan Kaufmann、IBM Redbooks、Packt, Adobe Press、FT Press、Apress、Manning、New Riders、McGraw-Hill、Jones & Bartlett、Course Technology 等。

更多信息，请访问：*http://oreilly.com/safari*。

## 致谢

首先，我必须感谢 Ionic 的整个团队，是他们创建了这个伟大的框架。首先是 Max Lynch 和 Ben Sperry，他们将疯狂的想法和激情转变为今天的 Ionic。我对它的未来充满期待。然后要感谢 Ionic 家族的其他成员：Adam Bradley、Mike Hartington、Brandy Carney、Dan Bucholtz、Tim Lancina、Alex Muramoto、Matt Kremer、Justin Willis 和 Katie Ginder-Voge，感谢你们能够抽时间回答我的问题，阅读本书稿件，以及在本书编写过程中提供指导，和你们合作非常愉快。当然我同样还要感谢 Ionic 团队的其他成员。

我诚挚地感谢本书的两位技术评审：Ray Camden 和 Leif Wells。由于你们的建议和细心指点，使得本书增色不少，非常感谢你们能花时间认真读完我的第一本书。

特别感谢 O'Reilly 公司的 Meg Foley，他是我的贴心、耐心的编辑。因为技术的不断更新，本书完成时间比预期长。同时要感谢我的经纪人，即 Waterside Productions 公司的 Margot Hutchison，是他让我有机会认识了 Meg Foley。

感谢我的朋友们，感谢你们在本书创作过程中给予我的鼓励。现在，我们可以在下次见面的时候谈论一些大事了，比如精酿啤酒或火箭发射。

最后，感谢我的妻子 Anita 和我的双胞胎儿女 Ben 和 Shira，感谢你们赐予我时间，支持我回到创作本书的计算机和工作中来。感谢你们所做出的牺牲。我以为我永远完不成这件事了，是你们给予了我信心。

# 第 1 章

# 混合移动 App

移动应用开发已经变成了开发者必备的核心技能之一。在过去十年，我们看到了移动设备的爆炸式增长（从手机、平板到可穿戴式产品），它们构成了一个完整的移动应用生态链。可以说我们处在了一个移动 App 的时代。但这样我们就必然面临一个新的挑战：如何创建移动 App？一般而言，一个开发者需要学习和掌握每一个平台下的语言：如果要编写 iOS 应用，那么要掌握 Objective-C 或者 Swift，如果要编写 Android 应用，则需要学习 Java。如果有一种方法可以只学习一种语言就能跨过多个平台下使用，那就好了。答案是肯定的：通过使用 Web 通用语言和某些神奇的框架，开发者能够在开发应用时“一次编码，部署到多个移动平台”。这就是所谓的混合移动 App，因为它是用 Web 技术开发的，但融合了移动设备的原生能力。

准确地说，什么是混合移动应用？和传统的原生移动应用不同，它不使用设备原生开发语言进行开发。混合 App 是用 Web 技术（HTML、CSS 和 JavaScript）开发的。实际上在你正在使用的移动设备上，就有许多 App 属于混合 App。

Ionic 框架是当今最流行的混合移动 App 框架。这个框架在 GitHub 上拥有着超过 26000 的 star 数，forked 数则超过了 5700 次。当这个框架的下一个大版本发布时，预计这个数字仍将随着混合移动开发者的增加而继续增长。

本书通过向导方式教你编写三个独立的 App 来演示如何编写 Ionic 2 应用程序。每个应用程序都会向你介绍 Ionic 框架的不同组件，同时让你了解 Ionic 生态系统。

在开始编写第一个应用程序之前，我们需要对 Ionic 的各个组成部分，以及本书中将会使用到的一些工具进行很好的了解。

## Ionic 框架是什么？

但到底什么是 Ionic 框架呢？简单来说，它是一个用 HTML、CSS 和 JavaScript 构成的用户界面框架，专门用于混合移动 App 的开发。除了用户界面组件，Ionic 框架也包含一个强大的命令行接口（CLI）和一套附属服务比如 Ionic View 和 Ionic Creator。在这本书中，我们会介绍每一部分。

Ionic 实际上融合了多种技术，它使得移动应用的编写变得更加轻松和快捷。整个技术栈的最上层是 Ionic 框架自身，它提供了应用程序的用户界面层。接下来一层是 Angular（完整的叫法是 AngularJS），这是一个极其强大的 Web 应用框架。这些框架的下面是 Apache Cordova，它允许 Web 应用程序调用设备原生能力并将 App 转换成原生 App。

这些技术融合在一起，使 Ionic 变成了一个创建混合 App 的强大平台。在本书中，将依次介绍这些技术。

## Ionic 2 有什么新特性？

若说 Ionic 2 是一次重大的升级，一点也不为过。除了 Ionic 框架自身的巨大改变，还有一个重大的变化就是底层技术的变化，即 Angular。有的东西尽管表面上看起来没有太大变化，但底层却发生了本质上的变化。Ionic 2 基本上是一个全新的框架。有许多组件的语法看起来与 Ionic 1 一样，但它们的实现已经是全新的了。

### Ionic（第 3 版）

在 2017 年 3 月，Ionic 公布了框架的第 3 版。新版本更像是一次传统意义的升级，而不是像 Ionic 1 到 Ionic 2 那样的重大升级。官方还宣称框架将继续沿用 Ionic 这个名字而不添加版本号。为了清晰，本书将使用 Ionic 2 来区别于 Ionic 1。

以下列出这个框架的主要改进：

*全新的导航*

彻底控制 App 的导航体验，不再局限于 URL bar。可以导航到任意视图中的任意页面，包括模式窗口、侧滑菜单以及其他的视图容器，同时保留 deep-linking 的能力。

*原生支持*

目前 Ionic 支持更多的原生功能，能轻松调用设备提供的全部能力，不需要到处搜索扩展插件和代码了。

*强大的主题*

通过全新的主题系统，马上就能轻松地将你的品牌颜色和设计进行匹配。

*材料设计*

为 Android App 提供完整的材料设计支持。

*Windows Universal App*

支持开发运行在 Windows Universal 平台上的 App。

当然，Ionic 的这些改进需要我们学习新版本的 Angular 和 TypeScript。我们将在后面的章节中介绍它们。

# 不同移动开发方式的比较

当你需要将你的 App 部署到某个移动平台时，有三种主要的方式供你选择，每种方式都有它们各自的优缺点，分别是原生移动 App、移动 Web App 和混合移动 App。为了对整个移动应用生态有一个了解，我们分别进行介绍。

## 原生移动 App

通常，当需要创建移动应用时，大部分开发者都会想到原生代码。要创建一个原生 App，开发者需要用不同的移动平台所支持的默认开发语言来编写代码，比如针对 iOS 设备的 Objective-C 或 Swift，针对 Android 的 Java，以及针对 Windows Universal 的 C# 或 XAML。

和别的方式相比，这种开发方式具有几个强大的优势。首先，开发工具和设备平台紧密绑定。开发者所使用的 IDE 是专门针对在该平台下编写移动 App 而配

置的：iOS 使用 Xcode，Android 使用 Android Studio。其次，因为是使用原生框架进行开发，所有的原生 API 和功能对于开发者都可用，而不需要使用任何其他的桥接方案。第三，App 的性能可以尽可能地优化。因为 App 运行在原生状态，没有中间层代码的性能开销。

原生移动 App 的主要缺点是开发只能使用某一种或几种语言。因为通常我们想将 App 部署到 iOS 和 Android 两种（也有可能还要加上 Windows）平台，你就需要熟悉两种不同的语言和 API。任何一方的客户端代码都无法共用，我们不得不重复编写代码。此外，维护多个代码库也是一种技术负担。

## 移动 Web App

当 iPhone 第一次上市时，根本没有第三方应用，也就无所谓什么 App 商店了。事实上，最早的第三方 App 版本仅限于移动 Web 应用而没有原生应用。现在当然不是这么回事了，但编写一个移动 Web App 仍然是一个不错的选择。这些 App 可以通过设备上安装的 Web 浏览器进行加载。尽管移动 Web 网站和移动 App 之间的界限越来越模糊，但这种技术仍然只使用 Web 技术来创建 App 并通过设备浏览器进行发布。

这种方式的优点是可以将我们的 App 部署到非常多的平台。不单是 iOS 和 Android 平台，别的平台也可以。你只需要考虑你的目标市场就可以了，这是一个重要的因素。因为直接访问了你的 Web 服务器，这种 App 相比较于本地 App 来说，复杂、缓慢的审批流程根本不是问题。如果你需要为 App 升级一个新特性或者解决某个 bug，方法非常简单，直接上传新包到服务器就行了。

但是，因为这些 App 直接运行在本地浏览器上，也会带来某些限制。首先，浏览器无法访问设备的全部能力。例如，浏览器无法访问设备的联系人信息；其次是 App 不可搜索。用户通常想通过设备上的 App 商店来搜索 App。此外，打开浏览器输入 URL 这样的动作也不是人人都喜欢。

## 混合移动 App

一个混合移动 App 某种程度上也是一种原生移动 App，它使用了一种原始的 Web 浏览器（我们通常称为 WebView 的浏览器）来运行 Web 应用。这种方式

在原生设备和 WebView 之间用了一个原生 App 容器来进行桥接。混合 App 有许多优点。和 Web App 一样，主要代码都能在多个平台下部署。它使用某种通用语言进行开发，代码库的维护更加轻松，如果你就是一个 Web 开发者的话，不需要学习一门全新的语言。和 Web App 不同的是，我们能够访问设备提供的完整能力，这通常是以某种插件系统的形式来进行的。

但是，这种方式也有一定的缺点。因为 App 仍然是一个 Web App，它的性能和容量受设备浏览器的限制。它的性能也会因设备而异。对于一些老旧设备，它们的移动浏览器性能非常差，从而使 App 的性能不太理想。尽管这种方案也是一个本地 App，但在 WebView 和设备原生能力之间仍然需要通过插件来进行通信。这会在你的项目中出现另一种依赖，同时这种方法不能保证 API 是可用的。最后，在 WebView 中，其他原生 UI 组件都是不可用的。你的 App 中所有的 UI/UX 都必须自己编写。

Ionic 框架使用的是混合 App。Ionic 团队花了大量的时间重写了 Web UI 组件，使它们看起来和原生组件外观没有区别。这个框架使用了 Cordova，通过它的插件库，只能访问少部分设备能力的问题已经被解决了。

# 理解 Ionic 的技术栈

现在，我们对移动应用开发方式有了一个大概的了解，接下来我们来深入解 Ionic 框架的工作原理。Ionic 应用由三个层次构成：Ionic、Angular 和 Cordova。

## Ionic 框架

Ionic 框架首创于 2013 年 11 月，它很快流行起来并不断发展。Ionic 基于 MIT 协议并在 Ionic 框架官网上提供下载（*http://ionicframework.com*）。

Ionic 框架的主要功能是提供了 Web App 开发中所不具备的 UI 组件。例如，在许多移动 App 中都会有一个 UI 组件 Tab Bar，但这个组件在原生的 HTML 标签中根本不存在。Ionic 框架扩展了 HTML 库并提供了一个这样的组件。这个组件是由 HTML、CSS 和 JavaScript 编写的，它的每个行为和外观都被重写了，和原生控件一模一样。

Ionic 还有一个 CLI 工具，用于轻松创建、编译和发布 Ionic 应用程序。我们会在本书中大量使用到它。

Ionic 平台还扩展出几个附属功能，包括一个在线的图形用户界面的构造工具，用于以图形化方式设计 Ionic App、打包及更新。尽管这些 Ionic 服务可以免费供开发者测试和使用，但如果是商用则需要按月付费。

Ionic 框架的主要目标是 UI 层，它通过集成 Angular 和 Cordova 的方式来提供接近于原生的体验。

## Angular

Ionic 技术栈的下一层是 Angular（正规的叫法是 AngularJS），这是一个由谷歌支持的开源项目。自从 2009 年开始发布以来，Angular 已经变成了最流行的 Web 应用框架。Angular 的目的是提供一个用于建造复杂的、单页面 Web App 的 MVW（model-view-whatever）框架。Ionic 团队决定利用 Angular 框架提供的强大功能，因此将它集成到了 Ionic 中。例如，Ionic 的定制 UI 组件其实就是 Angular 组件。Angular 遵循 MIT 协议，Angular 官网上提供了组件下载（*https://angular.io*）。

随着 Angular 2 的推出，框架也进行了大量改进。这个改变在 Angular 社区引发了一些争论，但关于这个框架的新特性的相关问题，有许多已经得到了解决。我们会在本书第 4 章详细介绍 Angular 2。

## Cordova

Ionic 框架技术栈中的最后一层是 Apache Cordova。Cordova 起源于 Nitobi Software 公司于 2009 年所开发的一个开源项目，能够利用 Web 技术来构建嵌入 WebView 的原生 App。在 2011 年，Adobe System 收购了 Nitobi（包括 PhoneGap 这个商标），这个项目不得不重新命名。尽管这个项目一直是开源的，但项目名字不是开源的。最终这个开源项目的版本被重命名为 Cordova（Nitobi 公司位置所在的街道）。PhoneGap 的创始人之一 Brian Leroux 说过，“PhoneGap 基于 Cordova。它们就好比 Webkit 和 Safari 的关系。”Adobe 仍然是 Cordova 的主要贡献者之一（主要贡献者还包含另外几个软件公司），它采用 Apache 2.0 协议。

Cordova 提供了 WebView 和设备原生层之间的接口。这个库提供了一个桥接框架，弥补这两个技术栈之间的空隙（这就是原名叫做 PhoneGap 的来由）。有许多功能是以插件系统的形式提供的，这种形式使得核心库保持较小规模。除了最主要的两个移动平台之外，Cordova 还支持其他移动平台，比如 Windows Phone、Blackberry 和 FireOS。完整的支持列表，请参考 *https://cordova.apache.org*。

除了库本身，Cordova 还拥有自己的命令行工具，用于搭建脚手架、编译和部署你的移动应用。Ionic CLI 是基于 Cordova CLI 构建的，我们将在本书中用到它。

## 进行 Ionic 应用程序开发的必备条件

为了开发 Ionic 应用程序，你必须具备一些没有在本书中包含的其他技能。你不一定是这些技术的专家，但为了更好地理解 Ionic 开发理念，你必须对这些技术有所了解，包括：

*懂得 HTML、CSS 和 JavaScript*

因为 Ionic App 是使用 HTML、CSS 和 JavaScript 编写的，你必须对如何综合利用这些技术创建 Web App 有大体上的理解。我们会使用 HTML 来搭建 App 的基本结构。CSS 会用来提供 App 的可视样式。最终，用 JavaScript 来编写 App 的逻辑和流程控制。

JavaScript 使用得会比较多一点，你必须熟悉它的语法和概念，比如变量作用域、异步调用和事件。

*理解 Angular 2*

除了理解基本的 HTML、CSS 和 JavaScript 之外，你还需要对构建 Web App 有一定的了解。在本书中，我们将主要使用 JavaScript，尤其是 Angular 2 编写 App。也就是说，我们会用到 ES6 和 TypeScript 来编写代码。你可能对这些内容有一点陌生，我们将在第 4 章进行一些基本的介绍，以便让你跟上进度。

*拥有一台移动设备*

毫无疑问，你还需要一台真实的移动设备，用于安装、测试你的 App。事实上，你需要为每个计划部署的平台准备至少一台设备。虽然 iOS 和 Android

SDK 提供了模拟器 / 虚拟机，让你能够测试 App 的外观和功能，但却取代不了真实设备。

## 小结

希望你已经对移动 App 的各种开发模式之间的差异，以及 Ionic 技术栈的构成有了一个比较好的理解，同时也对学习 Ionic 开发必须具备的条件有了更清晰的认识。

在下一章，我们将介绍如何针对 Ionic App 开发对你的计算机进行一些配置。

# 第 2 章

# 配置开发环境

使用 Ionic 框架进行开发的第一件事就是安装和设置 Ionic 所需的一些工具。这一章我们将带领你完成所有 Ionic App 开发所必需的组件的安装和配置过程。安装过程分成两个主要部分：基本的 Ionic 安装，以及指定平台 SDK 的安装。基本安装只包含编写第一个 Ionic App 工具的安装，然后通过浏览器来运行它。如果你是一个行动派并且想马上就开始使用 Ionic，那么这些就足够了。第二部分的安装是配置你的原生开发环境。尽管我们是用 Web 技术来编写 App，但仍然需要在我们的计算机上安装原生开发环境。这样才能使用模拟器，并能够在设备上部署和测试应用程序，最终将 App 提交到应用商店。

在整本书中，我们都会以命令行的方式来使用 Ionic CLI。在 MacOS 中，我们使用的是终端程序。我们建议在桌面上添加一个快捷方式或者将它拖到你的 Dock 栏上。如果你使用 PC 进行开发，建议你使用 Git Bash（我们在安装 Git 的时候也会安装 Git Bash），而不要使用系统默认的命令行提示符。两者使用的命令语句是一样的，因此你可以放心地照搬示例中的代码。

## 安装 Ionic 框架

本节将搭建基本的 Ionic 开发环境，编写我们第一个 Ionic 应用并在浏览器中运行它。你可能会奇怪为什么会在浏览器中运行我们的 App？别忘了，我们的 App 是用 Web 技术来编写的，因此把浏览器作为我们的第一个“目标平台”是

完全可以的。我们可以利用浏览器的调试工具，加快开发周期。在我的开发周期中，会尽可能避免在设备上的测试。

我们需要安装的组件有 4 个，见下表，你会看到需要安装的软件和对应的 URL。

| 工具 | URL |
| --- | --- |
| Node.js | nodejs.org |
| Git | git-scm.com |
| Ionic | ionicframework.com |
| Apache Cordova | cordova.apache.org |

## 安装 Node.js

Ionic 是基于 Node.js（通常简称为 Node）构建的。Node 是一个能够让你在浏览器之外运行 JavaScript 的平台。它允许开发者创建用 JavaScript 编写并几乎能在任何地方运行的应用程序。Ionic 和 Cordova CLI 都是用 Node 编写的。因为这个原因，我们首先需要安装的就是这个框架。

要安装 Node，请打开 Node 官网（*http://nodejs.org*），根据你的开发平台下载对应的安装包。如果已经安装过 Node 6.x，可以跳过这个步骤，直接使用 Node 6.x 版本。如果你还需要更新的 Node 版本，可以找一个 Node Version Manager 的东西，它允许你在不同的 Node 版本之间自由切换。

安装完 Node 后，打开终端并输入 **node -v**。这个命令会打印 Node 当前安装的版本号：

```
$ node -v
$ v6.9.2
```

如果安装不成功，请查阅文档。

你还应该确认 NPM（一个 Node 模块的包管理器）是否更新（注意：NPM 并不仅仅是“Node 包管理器”）。在你安装 Node.js 时，NPM 已经自动更新了。但如果你想看看 NPM 的安装版本，你可以使用下面的命令：

```
$ npm -v
```

```
$ 3.10.9
```

如果你需要更新 NPM 的安装，可以用这个命令：

```
$ npm install npm -g
```

当 Node 和 NPM 安装成功后，就可以安装 Git 了。

## 安装 Git

你可以自由选择任何版本控制方案（Perforce、SourceSafe 或者 Git），但是 Ionic CLI 是使用 Git 来管理模板的。另外，我发现 Windows 用户更容易通过 Git Bash 来使用本书中的示例进行学习。

打开 *http://git-scm.com*，单击下载按钮。打开安装包，依照默认方式进行安装。

安装完成后，打开终端进行测试。

在终端窗口中，输入 **`git --version`**，然后按回车键：

```
$ git --version
$ git version 2.8.4 (Apple Git-73)
```

Git 现在已经安装到我们的系统中了，接下来可以安装 Apache Cordova CLI。

## 安装 Apache Cordova CLI

尽管可以同时安装 Cordova 和 Ionic，但建议将它们分开来单独安装，因为同时安装时可能会出现一些问题。

安装 Cordova CLI 需要用到 Node 包管理器 (NPM)。要安装 Cordova CLI，打开终端窗口或者 Git Bash，输入下列命令：

```
$ npm install -g cordova
```

安装需要一段时间，这取决于你的网速。对于 MacOS 用户，在安装过程中可能会遇到一个需要管理员权限的问题。有两种选择：重新运行 `npm` 命令，但在命令前加一个 sudo 命令。这会让 Node 模块以 root 身份运行，也可以通过配置系统方式解决权限问题：

```
$ cordova -v
$ 6.4.0
```

安装完这些工具，最后来安装 Ionic CLI。

## 安装 Ionic CLI

和 Cordova CLI 的安装一样，Ionic CLI 也通过 NPM 进行安装。在终端窗口，输入下列命令：

```
$ npm install -g ionic
```

这个安装也需要一段时间才能完成。Ionic CLI 安装完成后，我们可以在终端输入 **ionic -v** 命令来检查：

```
$ ionic -v
$ 2.1.18
```

现在针对 Ionic 开发的基本安装已经完成了。但是，我们最终还要在设备模拟器或真实设备上测试我们的 App，需要简单看一眼这些工具的安装。在此之前，先来创建一个测试 App，以便在浏览器中预览我们的 App。

# 新建 Ionic 项目

Ionic CLI 提供了一个简单的命令，允许你创建一个 Ionic 项目：`ionic start`。这个 CLI 命令会在当前目录下创建一个基本的 Ionic 应用。Ionic 框架通过一系列初始模板来创建一个项目的脚手架。这些模板包含一个指定的模板、一个 GitHub 库、一个 Codepen，或者一个本地目录，指定的模板包括 *blank*、*sidemenu* 和 *tabs*，我们将在本书中进行介绍。现在，运行下列命令来创建一个 Ionic 项目：

```
$ ionic start testApp --v2
```

因为我们并没有指定开始模板，Ionic CLI 默认将使用 tabs 模板。CLI 会开始下载模板并配置各种组件，并询问你是否需要创建一个 Ionic.io 账号。就目前而言，我们可以忽略这一步，但在本书后续部分将介绍这个 Ionic 服务。当创建完成后，我们需要切换到由 CLI 创建的这个 *testApp* 目录：

```
$ cd testApp
```

我们先来看一看这个目录中都有什么内容。

## Ionic 项目的文件结构

项目目录中包含了大量文件和子目录，现在花点时间了解一下它们分别是什么，以及它们的用途。

| | |
|---|---|
| *src* | 这个目录包含了我们将会编写的App的代码。在Ionic 2之前的版本中，这个目录的名字是 *app* |
| *hooks* | 这个目录包含编译过程中 Cordova 会用到的脚本 |
| *node_modules* | Ionic 现在使用 NPM 作为它的模块管理系统，这里是它的支持库 |
| *resources* | iOS 和 Android 的默认图标以及启动图 |
| *platforms* | 这个目录包含每个安装过的平台的构建元素 |
| *plugins* | 这个目录包含了 Cordova 的插件 |
| *www* | 这个目录包含了 *index.html*，这个文件加上 *app* 目录中转译后的结果会用于启动我们的 Ionic App |
| *.gitignore* | 自动产生的默认的 gitignore 文件 |
| *config.xml* | Cordova 用这个文件来定义各种 App 相关的元素 |
| *ionic.config.json* | Ionic CLI 用来指定执行命令时的各种设置 |
| *package.json* | 这个项目中已安装的全部 npm 包的列表 |
| *tsconfig.json* | 用于定义编译项目时会用到 root 文件和编译器选项 |
| *tslint.json* | TypeScrpt 的 linter 规则 |

这是一个 Ionic 2 项目的标准结构。当我们添加了平台和插件，又会产生相应的子目录和文件。

**隐藏文件**

任何以点开头的文件在 Finder 中都是不可见的。

### 从 Ionic 1 到 Ionic 2 的改变

如果你用过 Ionic 1，你可能想知道有什么改变。首先，Ionic 不再用 Bower 作为它的包管理器，而是使用 NPM 来替代。但最大的不同在于，你不用在 *www* 目录中编写 App 代码了，而是在 *src* 目录中进行开发。

我们会在下一章介绍项目中的各种元素。现在，我们只需要在浏览器中测试我们的 App 及开发环境都可行即可。关于从 Ionic 1 迁移到 Ionic 2 的细节问题，请阅读附录 A。

### 在浏览器中进行预览

编写混合 App 的好处之一，是大量的开发和测试工作可以立即在浏览器中进行。在传统的原生应用开发中，你必须编译 App，然后要么在模拟器中运行，要么在设备上运行。Ionic CLI 中包含一个内置的命令，用于立即在浏览器中运行 App，输入这个命令：**`ionic serve`**。会启动一个简单的 Web 服务器，并打开你的浏览器，自动加载 App。同时它会在文件被保存时监听文件的改变并自动刷新浏览器。

#### 设置端口地址

大部分情况下，`ionic serve` 会提醒你选择一个 IP 地址。通常你只需要选择本地主机地址即可。如果 8100 端口被占用，你可以用 `--p` 加端口号的方式选择不同的端口。

我们会在浏览器中看到 Tab 风格的 Ionic App。Ionic 的 Tab 模板包含有多个窗口，我们可以在窗口间切换，并查看 Ionic 框架中的部分组件（见图 2-1）。

因为你是通过浏览器查看 App 的，所以仍然可以使用经常用到的那些开发者工具。

#### 浏览器选择

虽然你可以使用任何浏览器，但我仍然建议你使用 Google Chrome。尽管它和你在移动设备上运行的浏览器不是完全相同，但它们却很接近，这样无论你在桌面浏览器上或移动浏览器上进行测试都不会有什么问题。

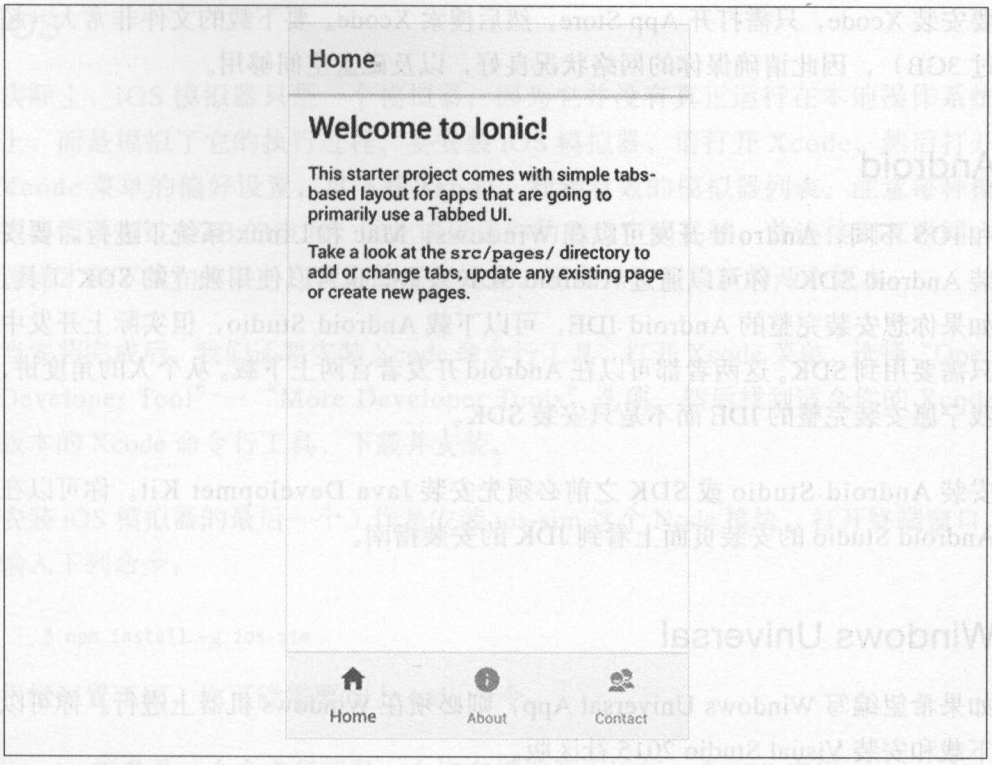



图 2-1：示例 App——基于 Tab 模板

# 安装平台工具

虽然我们已经拥有了开发环境，最终还是需要在模拟器或设备上继续开发，因此就需要安装原生 App 平台工具。本节内容比前面的安装内容稍微复杂，我们需要分别针对每一种平台进行安装。幸运的是，这是个一次性的过程，因此请花一点时间来完成这一节的内容。

当前，Ionic 正式支持的平台有 iOS、Android 和 Windows Universal。

## iOS

如果你准备针对 iOS 进行开发，你需要使用 Xcode 才能模拟运行和发布你的 App。Xcode 只能在 Mac 计算机上使用。尽管有一些办法可以绕开 Mac 这种硬性规定（PhoneGap Build 和 Ionic Package），但为了能够顺利完成开发，还是建议你至少拥有一台入门级的 Mac 计算机。

要安装 Xcode，只需打开 App Store，然后搜索 Xcode。要下载的文件非常大（超过 3GB），因此请确保你的网络状况良好，以及磁盘空间够用。

### Android

和 iOS 不同，Android 开发可以在 Windows、Mac 和 Linux 系统下进行。要安装 Android SDK，你可以通过 Android SDK 安装，也可以使用独立的 SDK 工具。如果你想安装完整的 Android IDE，可以下载 Android Studio，但实际上开发中只需要用到 SDK。这两者都可以在 Android 开发者官网上下载。从个人的角度讲，我宁愿安装完整的 IDE 而不是只安装 SDK。

安装 Android Studio 或 SDK 之前必须先安装 Java Developmet Kit。你可以在 Android Studio 的安装页面上看到 JDK 的安装指南。

### Windows Universal

如果希望编写 Windows Universal App，则必须在 Windows 机器上进行。你可以下载和安装 Visual Studio 2015 社区版。

在安装进程中，请选择“Tools for Cross Platform Development”以及“SDK for Windows Universal Apps”。

## 配置模拟器

安装完基本的移动 SDK，我们还没有完成安装过程。无论是 iOS 还是 Android，都需要创建对应的设备模拟器。这些模拟器允许你在电脑上运行虚拟的移动设备。我们可以用各种 OS 版本和设备类型的模拟器来快速测试我们的 App，而无需使用真正的物理设备。它们比起在浏览器上进行测试要慢，但能够测试和设备相关的特性，比如调用联系人数据库。

模拟器需要进行另外的安装和配置。让我们来看一下各个平台下都需要什么样的步骤。

## iOS

实际上，iOS 模拟器只是一个模拟器，因为它并没有真正运行在本地操作系统上，而是模拟了它的执行过程。要安装 iOS 模拟器，请打开 Xcode，然后打开 Xcode 菜单的偏好设置，再下载 Tab 页，找到有效的模拟器列表。注意每种模拟器需要超过 1GB 的空间，因此请确保你的磁盘空间足够，并连接到互联网，这样才能下载和安装。在我们的开发机上，通常只安装最新的两个版本。

当安装完成后，我们还要安装 Xcode 命令行工具。打开 Xcode 菜单，选择“Open Developer Tool”→“More Developer Tools”选项。然后找到适合你的 Xcode 版本的 Xcode 命令行工具，下载并安装。

安装 iOS 模拟器的最后一个工作是安装 ios-sim 这个 Node 模块。打开终端窗口，输入下列命令：

```
$ npm install -g ios-sim
```

根据配置不同，你可能需要加上 sudo 命令。

ios-sim 工具是一个命令行工具，允许在模拟器中运行一个 iOS 应用。

现在已经能够在 iOS 模拟器上运行我们的 Ionic App 了，马上就能看到。

## Android

在配置 Android 模拟器之前，我们必须安装并配置好 SDK 包。如果你只安装了 SDK，请在命令行中运行这句代码：[译注 1]

```
$ android sdk
```

这会打开单独的 SDK 管理器。这个工具允许你下载任意 Android 版本的平台文件。和 iOS 类似，我们建议你只下载最近两个版本的发布包和工具。

你需要选择下列选项：

---

译注 1：新的 SDK 不再支持 android 命令，请用 sdkmanager 命令替代。

- 指定目标版本的 Android 平台 SDK。
- Android 平台工具。
- Android SDK 构建工具版本 19.1.0 或以上。
- Android 支持库（在“Extras”下面）。

如果你使用的是 Android Studio，在欢迎界面选择 Configure，然后选中 SDK 管理器，选择和上面相同的组件进行安装。

当 SDK 以及对应的平台工具安装完后，我们就可以配置 Android 模拟器了。虽然我们可以在 Android Studio 中创建和配置虚拟 Android 设备，但那需要你创建一个有效的项目。因此，我建议你用命令行配置你的虚拟设备：[译注 2]

```
$ android avd
```

这会打开 Android 虚拟设备 (AVD) 管理器。打开后，选择“Device Definitions”标签页，然后选择一种已知设备配置。选择“Nexus 5 definition”，然后单击“Create AVD”按钮。这会打开一个新的窗口，显示你可以为虚机配置的各种配置和相应的细节：屏幕尺寸、所运行的 Android 版本等。当你设置完虚机后，单击 OK 完成配置。你可以拥有任意数目的虚拟设备。

### Android 模拟器

Android 模拟器的启动和运行非常慢。在最近的版本中这个问题有明显改善。当然，你的虚拟 Android 设备还可以有另外一个选择，就是 Genymotion。

## 配置你的设备

某些时候，你不得不在真正的移动设备上测试你的 App。要做到这一点，必须针对每个平台进行不同的设置。

译注 2：新的 SDK 不再支持 android 命令，请用 avdmanager 命令替代。

## iOS

虽然每个人都可以在iOS模拟器上测试他们的iOS App，但要在真机上进行测试则必须拥有付费的iOS开发者许可账号。以前，要激活你的iOS设备用于开发是一个复杂过程。幸运的是，最近的Xcode有一些改变，简化了这个过程。

1. 首先，直接将iOS设备连接上你的Mac。这个方法不能使用无线连接。然后需要创建一个临时的Xcode项目。在Xcode中，选择“File”→“New”→“Project”菜单。新建项目向导会打开，这时选择“Single View Application”模板。在下一个窗口，在“Project Name”栏输入“Demo”，然后单击“Next”。这里的设置并不重要，因为一旦配置好设备就会删掉这个项目。选择项目保存目录，然后单击“Create”。

   Xcode会显示项目的配置窗口。现在我们需要设置当前Scheme为我们的设备。可以通过单击Xcode窗口左上角附近的Scheme按钮来设置。

2. 将你的设备解锁并显示出Home屏，在Scheme下拉框中选择设备名称。这时会弹出一个警告“No Signing Identity Found”。不要让Xcode修复这个问题，我们需要手动解决它。

3. 在General设置中，打开Identity面板，[译注3]从下拉选项中选择你在团队中的名称（团队名很可能就是你的名字）。

   如果你不能看到你的名字，你需要将你的团队账号添加到Xcode中。要添加团队账号，在下拉列表中选择“Add Account”，Xcode偏好设置的Accounts页面将打开。输入和你的iOS开发者账号所对应的Apple ID和密码，然后单击“Add”。

4. 当Xcode完成登录并刷新列表，关闭Accounts窗口。从Team下拉列表中重新选择你新加的团队名称。

   Xcode会显示一个“No Matching Provisioning Profiles Found”警告。单击“Fix Issue”，Xcode会自动解决这个问题。

---

译注3：这是Xcode 5，实际上在最新的Xcode 8应该是在项目target设置的Gernal标签页，Signing一节下面的Team一栏。

为了配置 Provisioning Profile，Xcode 需要额外的信息和权限。你可以用默认选项进行回答。

5. 接下来检查这些步骤配置是否正确。单击 Xcode 左上角的“Run”按钮，并确认你已经将自己的 iOS 设备作为运行目标。几秒钟后，测试 App 就会在你的设备上启动了！

现在，回到命令行工具中来，我们需要安装另一个 Node 工具 ios-deply。在命令行中输入下列命令：

```
$ npm install -g ios-deploy
```

**在 El Capitan 上安装**

如果你使用的是 macOS 10.11 El Capitan，你可能需要在运行 `npm install` 时添加 `--unsafe-perm=true` 参数，否则安装将会失败。更多详情，请参考 GitHub 上的关于这个问题的描述。

更多帮助信息，请参考苹果官方文档。

## Android

配置一个 Android 设备和配置一个 iOS 设备的步骤截然不同。首先是在你的设备上开启开发者模式。因为每种 Android 设备的 UI 界面都不一样，只能描述一下大致步骤：

1. 打开设置，找到关于本机选项。

2. 应当能够看到一个 Build 编号，要开启开发者模式需要单击它 7 次。在这 7 次单击中，设备会提示你还剩下几次单击。

3. 当单击完毕，回到设置列表，你会看到一个新的开发者选项。

如果你无法开启设备的开发者模式，请翻阅设备的用户手册。然后为了能够部署我们的 App，需要开启 USB 调试。在开发者选项界面，找到 USB 调试选项并开启它。

现在你的 Android 设备应该可以用于开发了。当你将设备第一次连接到电脑时，可能会有一个确认配对的提示。

## 添加移动平台

尽管 Ionic CLI 会为我们的 App 创建脚手架，但我们还是要添加目标移动平台。为了在模拟器和设备上运行我们的 App，必须安装对应的平台。打开终端窗口，确认当前工作目录就是你的 Ionic 项目目录。相应的 Ionic CLI 命令是 `ionic platform add [platform name]`。

例如要添加 Android 的项目文件：

```
$ ionic platform add android
```

要添加 iOS 的项目文件：

```
$ ionic platform add ios
```

添加 Windows 的项目文件：

```
$ ionic platform add windows
```

默认地，如果你是在 Mac 下执行这个命令的话，iOS 平台就已经添加了，因此你不需要手动安装 iOS 平台。这个命令会生成指定平台的依赖文件。

## 在模拟器上测试

每当将一种移动平台添加到我们的 Ionic 项目，就可以在这个平台的模拟器上测试我们的 App。要在模拟器上运行 App，使用下列命令：

```
$ ionic emulate [platform]
```

这个 Ionic CLI 会将 App 编译到指定平台模拟器上。在编译过程中在终端可以看到大量的输出。当编译完成后，模拟器自动启动并运行你的 App。

如果你需要指定要模拟的设备，在命令后添加 `--target` 参数。例如，若要模拟一个 iPad Air 设备，我们可以用：

```
$ ionic emulate ios --target="iPad-Air"
```

若要查看全部 iOS 设备类型的列表，可以用：

```
$ ios-sim showdevicetypes
```

若要查看 Android 设备列表，需要参考 AVD 管理器的设备名称。

当模拟器启动并运行起来后，你可以再次使用 `emulate` 命令而无需关闭模拟器。每当文件修改后，用这个命令比关闭模拟器然后又重启它更节省时间。因为没有必要让模拟器每次都去重启操作系统。

Ionic CLI 还有另外一个非常强大的功能，允许实时重新加载 App，也就是 `--livereload` 参数。当你在使用模拟器开发时，这个功能能够极大地节省你在 Ionic 1 时浪费的时间。但是，新设备的安全特性将这个功能禁用了，解决方案目前还不得而知。

你还可以将控制台日志输出到终端窗口，这样你读起来就更方便了（关于这个特性，请参考：*http://blog.ionic.io/live-reload-all-things-ionic-cli/*）：

```
$ ionic emulate ios -l -c

$ ionic emulate android -l -c
```

## 在设备上测试

尽管模拟器很好用，但最后我们还是要将 App 安装到物理设备上。Ionic CLI 让这个过程变得非常简单，只需要使用 `run` 命令即可。在你的终端窗口中，输入 `ionic run platform`。

Ionic CLI 开始编译要安装到你的手机上的 App。编译完成后，它会自动将 App 安装到当前连接的手机。需要注意的是，每次安装都会覆盖上一次的安装。

当然对于 iOS 发布，必须事先安装好 ios-deploy 这个 Node 模块，否则你可能不得不用 Xcode 来手动安装了：

```
$ ionic run ios -l -c
```

如果你是在 Windows 机器上开发 Android 的，你可能需要下载并安装和你手机相匹配的 USB 驱动。请浏览 Android Studio 官网（*https://developer.android.*

*com/tools/extras/oem-usb.html*），以查看是否有符合你手机的驱动程序。如果你是在 MacOS 上发布到设备则不需要任何驱动程序：

```
$ ionic run android -l -c
```

如果没有找到任何设备，Ionic CLI 会将 App 部署到对应平台的模拟器或虚拟机上。

## 小结

本章讨论了创建 Ionic 开发环境的每个细节。我们还编写了第一个测试 App，先在本地浏览器和模拟器中进行预览，最后安装到设备上运行。

# 第 3 章

# 理解 Ionic 命令行界面

在开发 Ionic App 时，我们要用到的一个重要工具，就是 Ionic 命令行界面即 CLI。在我们配置开发环境时曾经简单介绍过这个工具，但在这一章，我们会介绍这个工具提供给我们的许多功能。

首先，如果你还没有安装 Ionic CLI，需要用 NPM 安装它。如果你使用 Mac 计算机，请打开终端程序。如果是 Windows 用户，请打开 Git Bash（或者其他命令行工具）。然后输入以下命令：

```
$ npm install -g ionic
```

这会安装 Ionic CLI 的最新版本。Ionic CLI 和 Ionic 1 项目是完全向下兼容的，如果你已经有部分工作是在 Ionic 1 下开发的，那也没有问题。

为了能够正常安装，macOS 用户可能需要在 `NPM` 命令前添加 sudo 命令。

一旦安装好 Ionic CLI，我们可以通过创建一个测试 App 来测试安装是否正确：

```
$ ionic start myApp [template name] --v2
```

这个命令会使用我们在命令行中指定的模板来创建一个新的 Ionic App 并保存在 *myApp* 目录下。先来看一下我们可以使用的模板都有哪些。

Ionic 当前支持三种模板：*blank*、*sidemenu* 和 *tabs*。如果不指定 template 参数，默认将使用 tabs 模板（见图 3-1）。

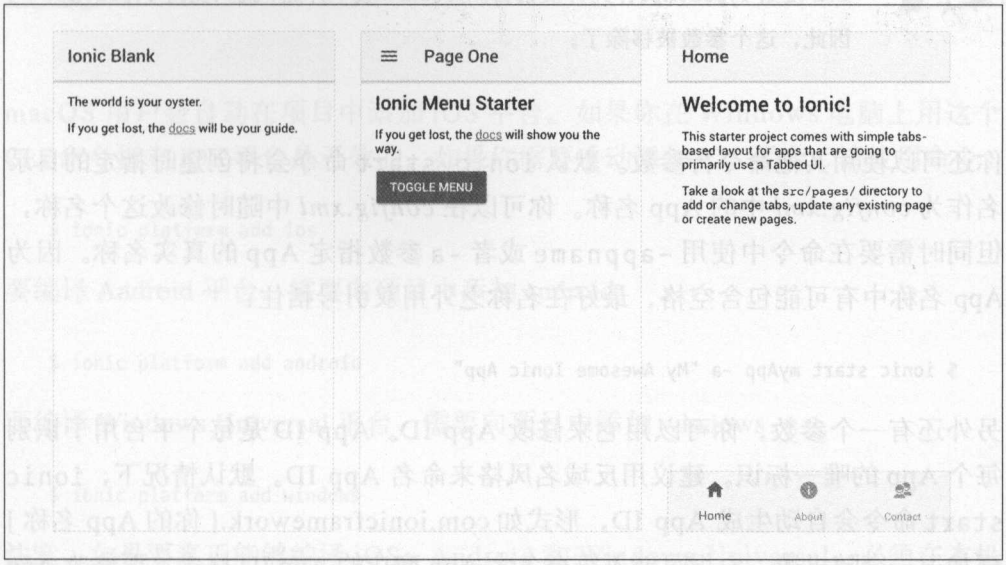



图 3-1：Ionic 的模板：blank、sidemenu 和 tabs

除了这三个命名的模板，还可以通过 URL 的形式指定自己的模板，可以将这些模板放在 GitHub、CodePen 或者其他网络地址上。事实上，这些命名的模板也只是位于 GitHub 上的代码库的别名而已。这里有一个用来自 GitHub 上的模板来创建 App 的例子：

```
$ ionic start myApp https://github.com/driftyco/ionic2-start-blank --v2
```

如果你不想将模板放到网上，也可以指定一个指向模板文件的相对路径或绝对路径的本地地址。

因为 Ionic CLI 支持 Ionic 1 和 Ionic 2，我们需要告诉它创建哪一个版本的 App，也就是 `--v2` 参数。通过在命令行指定这个参数，Ionic CLI 会用框架的第二版来创建 Ionic App 的脚手架。

```
$ ionic start myIonic2App --v2
```

**--ts 参数**

如果你用过 Ionic 2 之前的版本，你可能会知道 `--ts` 参数，这个参数用于告诉 CLI 使用 TypeScript 作为开发语言。从 beta 8 版本开始，就只支持 TypeScript 了。因此，这个参数被移除了。

你还可以使用其他命令行参数。默认 `ionic start` 命令会将创建时指定的目录名作为 *config.xml* 中的 App 名称。你可以在 *config.xml* 中随时修改这个名称，但同时需要在命令中使用 `-appname` 或者 `-a` 参数指定 App 的真实名称。因为 App 名称中有可能包含空格，最好在名称之外用双引号括住：

```
$ ionic start myApp -a "My Awesome Ionic App"
```

另外还有一个参数，你可以用它来修改 App ID。App ID 是每个平台用于识别每个 App 的唯一标识。建议用反域名风格来命名 App ID。默认情况下，`ionic start` 命令会自动生成 App ID，形式如 com.ionicframework.[ 你的 App 名称 ] 再加上一个随机数。它很可能不符合各大 App 商店的 App ID 格式。要修改 App ID，你可以在 `--id` 或 `-i` 参数后指定包名。这将修改 *config.xml* 文件中位于 widget 节点下的 id 属性：

```
$ ionic start myApp -i com.mycompany.appname
```

最后一个参数是 `--no-cordova` 或 `-w` 参数。这会告诉 `ionic start` 在创建项目时不要包含任何 Cordova 元素。你可能好奇为什么要这样做，难道我们不要创建移动 App 了吗？难道在编写 App 的过程中我们不需要 Cordova 了吗？答案是要。但是，当你只想用 Ionic 来创建移动 Web 应用时，你可以不使用 Cordova。另外一种可能是，你可能想将 Ionic 作为一个 UI 框架用于创建一个 Electron 桌面 App（*https://electron.atom.io/*）。在这两种情况下，可能要用到这个参数。

# 指定编译平台

当搭建好基本的 Ionic 应用程序框架之后，接下来需要添加我们准备编译的目标平台。这个命令是：

```
$ ionic platform add [platform name]
```

**常见错误**

如果你在刚刚创建的项目的目录之外运行这个命令，会发生一个错误。在运行其他 Ionic CLI 命令之前，请记得使用 `cd [app name]` 命令。

macOS 用户会自动在项目中添加 iOS 平台。如果你在 Windows 电脑上用这个 CLI 命令指定 iOS 平台是无效的。如果你需要手动添加 iOS 平台，请使用命令：

```
$ ionic platform add ios
```

要编译 Android 平台，需要向项目中添加 android：

```
$ ionic platform add android
```

要编译 Windows Universal 平台，需要向项目中添加 windows：

```
$ ionic platform add windows
```

注意，如果要真正能够编译 iOS、Android 和 Windows Universal，必须在本机上安装对应的 SDK。ionic platform add 命令的作用仅仅配置调用这些 SDK 的本地项目文件。

如果需要从项目中移除某个平台，你可以用：

```
$ ionic platform remove [platform name]
```

在安装某个插件或某次升级过程中，偶尔会出现错误。通常的解决方法是，重新将它安装到项目中。

## 管理 Cordova 插件

在添加编译平台之后的第一件事情，通常是安装 Cordova 插件。虽然在下一章我们也会讨论 Cordova 插件的使用和 Ionic Native，但这里先介绍几个基本的命令：

```
$ ionic plugin add [plugin id]
```

通常，插件 ID 就是 NPM 域名。例如，`cordova-plugin-geolocation`。但是它也可以是一个本地目录路径或者 GitHub 库。

要移除已安装的插件，使用命令：

```
$ ionic plugin rm [plugin id]
```

如果你需要查看项目中已安装的插件列表，可以用：

```
$ ionic plugin ls
```

# Ionic 生成器

尽管 Ionic CLI 会通过 `ionic start` 命令搭建 App 的脚手架，你仍然可以添加新的页面、组件、提供者、管道和其他东西来扩展你的 App。`generate` 命令允许你快速创建一个你所指定的元素的模板代码：

```
$ ionic g [page|component|directive|pipe|provider|tabs] [element name]
```

例如，如果你想在 App 中新增一个页面，命令是：

```
$ ionic g page myPage
```

Ionic CLI 会在我们的 app 目录中创建一个新目录，并创建对应的 HTML、CSS 和 TS 文件。

### 命名规范

Ionic 2 使用 kebob-casing（烤肉串命名法）来命名文件名（例如，*my-about-page.html*）和 CSS 类（例如，*.my-about-page*）。对于 TypeScript 中的 JavaScript 类则采用 Pascal 命名法（例如，*MyAboutPage*）。

# 预览你的 App

通常，开发初期可以在浏览器中进行浏览和测试。尽管 Ionic CLI 让你在模拟器或真实设备上运行 App 变得极其简单，但使用浏览器的好处是让你能够在最熟悉的环境中预览和调试 App。

*Ionic serve* 命令会启动一个本地开发服务器，自动载入你的 App 到浏览器中。此外，这个命令会使用一个 LiveReload 观察器，在你修改 App 的同时，自动重新加载浏览器中的内容，不需要手动刷新（见图 3-2）。

```
$ ionic serve
```

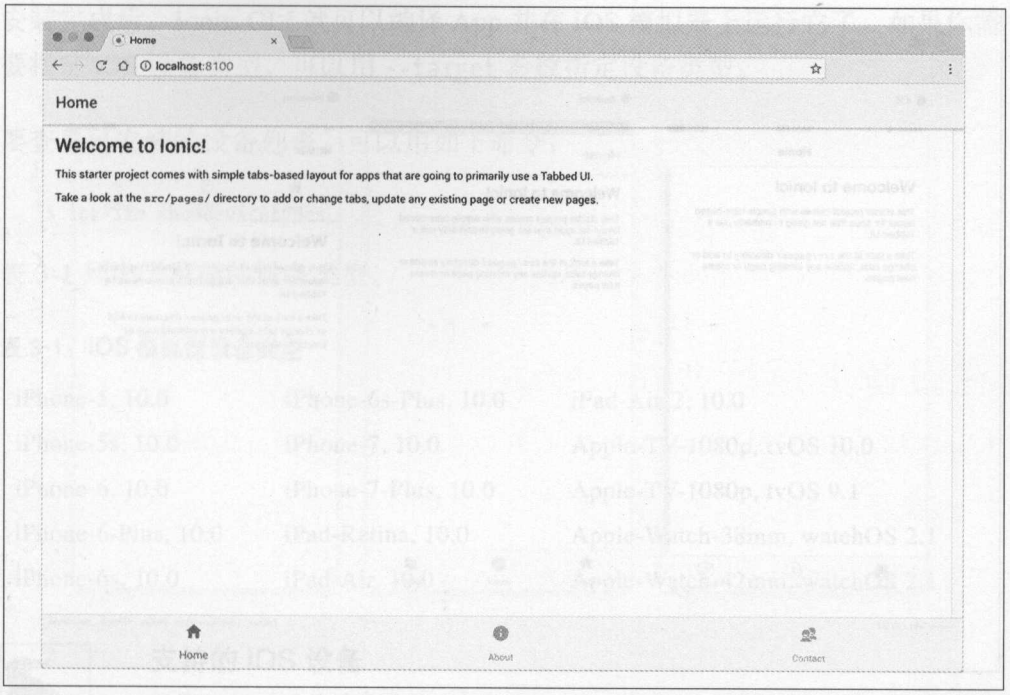



图 3-2：运行在浏览器中的 Ionic tabs App

## Ionic lab 命令

在 `ionic serve` 中添加 `--lab` 参数，你的 App 会在同一个浏览器窗口中显示 iOS 框架、Android 框架和 Windows 框架。这个功能允许你查看各个平台之间的差异。完整的命令如下（见图 3-3）：

```
$ ionic serve --lab
```

在这种模式下，每个平台下会运行一个 Ionic App 实例。因此所有平台相关的 CSS 或 JavaScript 都会被执行。在开发初期这个功能确实能节省不少时间，但却不能替代在真机上进行的测试。

# 指定 IP 地址

如果你需要指定 LiveReload 服务器运行的 IP 地址，可以在 `--address` 参数后加上 IP 地址：

```
$ ionic serve --address 112.365.365.321
```

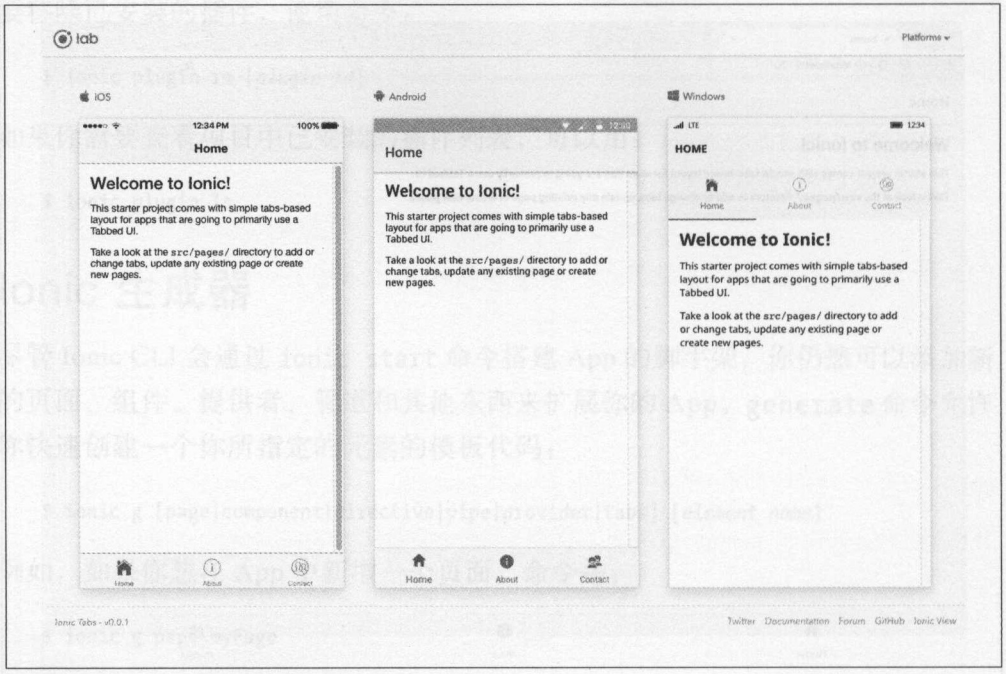



图 3-3：以 --lab 参数运行 ionic serve

# 模拟运行 Ionic App

`ionic emulate` 命令会编译 Ionic App 并在指定的模拟器或虚拟机上加载并运行 App：

```
$ ionic emulate android
$ ionic emulate ios
$ ionic emulate windows
```

模拟器可用于测试 App 中有可能调用到的一部分真机特性。在使用模拟器预览和运行 App 时，在加载 App 时可能会需要一定的时间。

## 模拟 iOS 设备

为了让 Ionic CLI 能够和 iOS 模拟器进行通信，还需要安装一个 Node 包。如果你之前没有安装过 ios-sim 包，那么现在就需要安装它了：

```
$ npm install -g ios-sim
```

安装完成后，Ionic CLI 就可以编译 App 并在 iOS 模拟器上运行它了。如果你需要指定 iOS 设备类型，可以用 `--target` 参数指定设备类型。

要查看已安装的设备列表，可以用如下命令：

```
$ ios-sim showdevicetypes
```

表 3-1 列出了可能的设备类型。

表 3-1：iOS 模拟器设备类型

| | | |
|---|---|---|
| iPhone-5, 10.0 | iPhone-6s-Plus, 10.0 | iPad-Air-2, 10.0 |
| iPhone-5s, 10.0 | iPhone-7, 10.0 | Apple-TV-1080p, tvOS 10.0 |
| iPhone-6, 10.0 | iPhone-7-Plus, 10.0 | Apple-TV-1080p, tvOS 9.1 |
| iPhone-6-Plus, 10.0 | iPad-Retina, 10.0 | Apple-Watch-38mm, watchOS 2.1 |
| iPhone-6s, 10.0 | iPad-Air, 10.0 | Apple-Watch-42mm, watchOS 2.1 |

**支持的 iOS 设备**

尽管 ios-sim 命令会列出 Apple TV 和 Apple Watch，但 Apache Cordova 和 Ionic 并不支持这些平台。

## 模拟 Android 设备

要在 Android 模拟器中模拟运行 Ionic App，首先需要手动创建一个供模拟器使用的 Android 设备（AVD）。如果你在上一章中没有完成这个环节，请使用如下命令：

```
$ android avd
```

这会打开 AVD 管理器，这是一个用于创建、管理各种 AVD 的工具。一旦你创建了一个 AVD，Ionic CLI 就能够启动 Android 模拟器并运行你的 App 了。模拟器的启动过程需要较长时间。

要指定具体的 Android 设备，可以用 `--target=NAME` 参数来使用你所创建的设备来运行 App，否则使用默认的模拟器。

**性能问题**

如果你正在使用 Android 模拟器，可以改善性能的技巧之一就是不要关闭模拟器，保持它的运行，而仅仅是重新加载 App。

尽管模拟器的性能已经进行过优化，但许多开发者更愿意使用 Genymotion（*https://www.genymotion.com/*）来代替标准的 Android 模拟器。

## 在设备上运行 Ionic App

Ionic CLI 也能够编译你的 Ionic App，并让它能够运行在经过正确配置的设备上：

```
$ ionic run [platform name]
```

如果当前电脑没有连接任何物理设备，Ionic CLI 会尝试部署到对应平台的模拟器上。

如果要部署到一台 iOS 设备上，你必须安装另外一个 Node 模块 ios-deploy：

```
$ npm install -g ios-deploy
```

为了能够部署成功，你的设备必须经过配置。如果使用的是 Android 设备，你只需要将它设置为开发模式即可。

## 输出日志

`emulate` 和 `run` 命令都支持将控制台日志和服务器日志重定向到 Ionic CLI。要开启控制台日志，可以用 `--consolelogs` 参数或者简写的 `--c` 参数。如果你想查看服务器日志，可以使用 `--serverlogs` 参数或者简写的 `--s` 参数。

## CLI 的信息

如果你查看 Ionic CLI 的全部状态和它的支持工具，请使用：

```
$ ionic info
```

在我的计算机上，会看到如下显示：

```
Your system information:
Cordova CLI: 6.5.0
Ionic CLI Version: 2.1.18
Ionic App Lib Version: 2.1.9
ios-deploy version: 1.9.0
ios-sim version: 5.0.13
OS: macOS El Capitan
Node Version: v6.9.2
Xcode version: Xcode 8.0 Build version 8A218a
```

在 debug 某个问题时，这些信息是很有用的。

## 小结

本章我们学习了核心的 Ionic CLI 命令，在开发过程中你会经常用到这些命令。除此之外还有一些其他的命令和参数。要查看完整的命令列表，请使用 ionic --help。

概括一下本章介绍的重要命令：

1. 用 `ionic start` 命令创建 Ionic App 的原始脚手架。
2. 用 `ionic platform` 命令管理项目的移动平台。
3. 在浏览器上预览 App，以及用 `ionic run/emulate` 命令在模拟器和设备上预览 App。

# 第 4 章

# Angular 和 TypeScript 基础

当配置好基本的 Ionic 开发环境之后，我们可以来看一下构成 Ionic 技术栈的一个关键技术，即 Angular。就像 Ionic 使用了 Apache Cordova 作为通向原生移动平台的桥梁一样，Ionic 用 Angular 作为界面层的基石。从一开始，Ionic 框架就构建于 Angular 框架之上。

## Angular 2 是什么？

Angular 2 是 Google 下一代强大的 MV* 框架。在 2014 年 10 月 ngEurope 大会上介绍了 Angular 的这个新版本。Angular 团队表示这个版本将会是一个重大的回归。从很多方面来说，新的 Angular 是一个全新框架，只是借用了原来的名字和名义而已。这种表示也引起了许多对该框架未来的担忧和不确定性。Ionic 社区和 Ionic 团队自身也存在这样的问题。哪些改变才是必要的？原有的 Ionic 开发者需要重新学习多少内容才能继续使用 Angular？

随着冲击力的消失，表现得越来越明显的一点是，Angular 框架的演变使它变得更好。这个框架变得愈发快捷、简单和强大，更容易学习。Angular 团队进行了一次大胆的尝试，他们看到了 Web 的未来而不是 Web 的过去。因此他们决定拥抱许多最新的和重要的 Web 标准，并用下一代的 JavaScript 来开发它。

在 2016 年的 ngConf 大会上，Angular 团队发布 RC 1 版本，这个框架将只使用 Angular 而不是 Angular 2 命名。但在本书中，我们还是引用 Angular 2 这个名称。

我们来看一看 Angular 的这些改变。

## 组件

从 Angular 1 到 Angular 2 的一个最大改变就是不再依赖 scope、controller 或者某种程度的 directive 了。Angular 团队使用一种基于组件的方法构建元素及其相关逻辑。使用传统框架开发过 App 的人对这种模式非常熟悉。可以这样说，我们过去使用的开发平台是用来看物理论文的，而不是用来开发 App 的。

这是一个 Angular 组件的例子：

```
import { Component } from '@angular/core';

@Component({
  selector: 'my-first-component',
  template: `<div>Hello, my name is {{name}}.
  <button (click)="sayMyName()">Log my name</button></div>`
})

export class MyComponent {
  constructor() {
    this.name = 'Inigo Montoya'
  }
  sayMyName() {

    console.log('Hello. My name is ',this.name,'.↵
                You killed my father. Prepare to die.')
  }
}
```

这和 Ionic 创建自己的组件库是一样的方式。事实上，你可以用自己的定制组件随意扩展 Ionic 应用程序，没有任何限制。

为了更加清晰，我们来看一看这些代码。

首先，第一句代码从 Angular 库中导入 `Component` 模块。在 Angular 中，这是通过依赖注入来进行的。在 Realse Candidate 1（RC1）中，Angular 团队将库拆分成更小的模块，同时在库中取消“2”的后缀。

接着，我们用 `@Component` 修饰符为代码添加一些元数据给编译器。我们决定使用一个自定义的 HTML 选择器。这样，只要我们一使用 `<my-first-component>/<my-first-component>` 标签，对应的模板就会插入到 DOM 中。模

板有两种风格：一种是像上面代码中一样的行内风格；另一种是外部引用。如果你的模板代码为了阅读方便需要排列成多行，请使用反引号（`）而不是单引号（'）括住模板代码。在这一章后续内容中我们会介绍更多关于模板的内容。

在修饰符之后，我们导出了类的定义，即 MyComponent。在类的构造函数中，我们将 name 变量赋值为 "Inigo Montoya"。和 Angular 1 及普通的 JavaScript 不同的是，Angular 2 对变量的作用域有更严格的限制。

最后，这个类还有一个公共方法 sayMyName，这个方法会向控制台输出一段文字。在使用 Ionic 和 Angular 2 的过程中，你会越来越熟悉和适应这种创建组件和页面的新方法。

## 输入

由于 Angular 2 是基于组件模型构建的，它需要有一种将信息传递到组件的机制。这是由 Angular 的 Input 模块来负责处理的。我们来看一个简单的组件 <current-user> 的例子，为了使用这个组件，需要一个 user 参数。这个标签使用起来如下所示：

```
<current-user [user]="currentUser"></current-user>
```

而这个组件的定义如下所示：

```
import { Component, Input } from '@angular/core';

@Component({
    selector: 'current-user',
    template: '<div>{{user.name}}</div>'
})

export class UserProfile {
    @Input() user;
    constructor() {}
}
```

在类的定义里面，有一个 @input 绑定到 user 变量。通过这种绑定，Angular 会传递 currentUser 变量给这个组件，从而让模板渲染出 user.name 值。

这就是 Ionic 组件强大的地方。我们可以用这个例子中的机制来进行数据的传递和参数的设置。

## 模板

模板是一个 HTML 片段，由 Angular 将指定元素和属性组装在一起构成的动态内容。从大部分内容上看，Angular 2 的模板系统没有太多改变。

### {}: 渲染

```
<div>
Hello, my name is {{name}}.
</div>
```

但是，和 Angular 1 不同，数据绑定只有一种方式。这样，生成的事件监听器的数量减少了，性能却提高了。

### []: 绑定属性

当组件需要解析和绑定一个变量时，Angular 会用 [] 语法。在本章讲到输入的时候曾经接触过它。

如果我们的组件中有一个 `this.currentColor`，要将这个变量传递给组件，Angular 会保证这个值始终是最新的值：

```
<card-header [themeColor]="currentColor"></card-header>
```

### (): 事件处理

在 Angular 1 中，我们会使用自定义指令来监听用户事件，比如元素的单击事件（有点像 `ng-click`）。Angular 2 中使用了一种更简单的方法，只需要将你想监听的事件用圆括号括住，然后赋给组件的方法即可：

```
<my-component (click)="onUserClick($event)"></my-component>
```

### [()]: 双向数据绑定

默认地，Angular 不会创建双向数据绑定。如果你需要用到双向数据绑定，新的语法也很简洁，将属性绑定语法和事件绑定语法结合起来：

```
<input [(ngModel)]="userName">
```

组件的 `this.userName` 将始终和输入值保持同步。

### *：星号

在某个指令前使用星号会告诉 Angular 将我们的模板以指定的方式进行处理。和渲染模板相反，它会首先对其应用 Angular 指令。例如，ngFor 会把我们的 <my-componet> 变成 for each item in items，同时它不会把最初的 <my-componet> 当成模板来渲染：

```
<my-component *ngFor="let item of items">
</my-component>
```

## 事件

Angular 2 中的事件在模板中使用圆括号进行标记，并触发某个组件类中的方法。例如，假设我们使用这个模板：

```
<button (click)="clicked()">Click</button>
```

同时该组件的定义是：

```
@Component(...)
class MyComponent {
  clicked() {
  }
}
```

当按钮被单击时，这个组件中的 clicked() 方法将被调用。

另外，在 Angular 2 中，事件就像普通的 DOM 事件。它们能够向上冒泡并向下广播。

如果我们需要访问这个事件对象，可以在事件回调方法中简单地传递一个 $event 参数：

```
<button (click)="clicked($event)"></button>
```

这个组件类应当变成：

```
@Component(...)
class MyComponent {
  clicked(event) {
  }
}
```

### 自定义事件

如果你的组件需要向其他组件广播自定义事件怎么办？ Angular 2 中有一个很简单的办法。

在我们的组件中，我们可以导入 Ouput 模块和 EventEmitter 模块。然后我们就可以用 @Output 修饰符定义自己的自定义事件 userUpdated 了。这个事件是一个 EventEmitter 实例：

```
import {Component, Output, EventEmitter} from '@angular/core';

@Component({
  selector: 'user-profile',
  template: '<div>Hi, my name is </div>'
})
export class UserProfile {
  @Output() userDataUpdated = new EventEmitter();

  constructor() {
    // 修改 user
    // ...
    this.userDataUpdated.emit(this.user);
  }
}
```

当我们想触发事件的广播时，你可以在自定义事件上调用 emit 方法并传入一个参数，这个参数将随事件一起传递。

现在，我们可以在 App 中使用这个组件了，并可以绑定 user-profile 发出的事件了：

```
<user-profile (userDataUpdated)="userProfileUpdated($event)"></user-profile>
```

当我们将 UserProfile 组件导入我们的另外一个组件后，它就能监听到 userProfileUpdated 事件的广播了：

```
import {Component} from '@angular/core';
import {UserProfile} from './user-profile';

export class SettingsPage {
  constructor(){}

  userProfileUpdated(user) {
    // 处理事件
  }
}
```

### 生命周期事件

Angular App 和它的组件都有生命周期事件，允许开发者访问生命周期中的每个关键环节。这些事件通常和它们的创建、渲染、销毁相关。

### NgModule

Angular 团队通过 NgModule 函数重新实现了 App 的引导方式。这个工作直到 Angular 的 RC 版发布才完成，因此这个功能的出现可能让人有点意外。`@NgModule` 使用了元数据对象告诉 Angular 如何编译和运行模块代码。另外，`@NgModule` 允许你将所有的依赖进行前置声明，而不用在 App 中声明多次：

```
import { NgModule }      from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { AppComponent }  from './app.component';

@NgModule({
  imports:      [ BrowserModule ],
  declarations: [ AppComponent ],
  bootstrap:    [ AppComponent ]
})

export class AppModule { }
```

这段代码是一个基本的 `app.module.ts` 文件的例子，它使用了 `BrowserModule` 模块，这样 Angular App 才能运行在浏览器中，然后声明并启动 AppComponent。

这个模块最终会被 *main.ts* 文件所用，并进行真正的引导过程：

```
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
import { AppModule } from './app.module';

const platform = platformBrowserDynamic();

platform.bootstrapModule(AppModule);
```

这段代码初始化了 App 即将运行的平台，然后用这个平台引导你的 AppModule。Ionic starter 模板会自动为你创建必要的模块。

NgModule 的另一个好处是允许我们使用 AoT 编译器（预先编译器），这种编译器能让 App 更快。

### 组件初始化事件

当组件被创建时，它的构造函器数会被调用。在构造器中，我们必须进行组件在构造时必须进行的初始化。但是，如果我们的组件需要从子组件获得某些信息或属性，我们可能无法访问这些数据。

Angular 提供了一个 `ngOnInit` 事件，这个事件会在组件初始化真正完成时触发。我们的组件可以在框架调用这个方法之前等待，然后就可以在组件中使用和解析所有的属性了。



### 组件生命周期事件

除了 *ngOnInit* 事件，组件还有其他几个生命周期事件：

*ngOnDestroy*

这个方法在组件被框架销毁前调用。你可以在这里取消对 Observable 的订阅，解绑事件处理器以免内存泄漏。

*ngDoCheck*

可以通过这个方法自定义变化监听。

*ngOnChanges(changes)*

在检查循环中，当组件的某个绑定值发生改变时，该方法会被调用。这个方法的参数是一个对象，格式如下：

```
{
    'prop': PropertyUpdate
}
```

*ngAfterContentInit()*

和 `ngOnInit` 不同，这个方法在内容被渲染之前调用，这个方法会在内容第一次渲染到视图时调用一次。

*ngAfterContentChecked*

这个方法在 Angular 检查完外部内容的绑定值后调用，这些内容会渲染到视图上。

*ngAfterViewInit*

这个方法会在 Angular 创建组件的视图后调用。

*ngAfterViewChecked*

这个方法是组件初始化过程的最后一个方法，当所有组件视图中的数据绑定被解析后调用一次。

### Ionic 事件

尽管你可以使用 Angular 事件，但仍然建议你使用 Ionic 事件。表 4-1 列出了它们的调用时机。

表 4-1：Ionics 事件描述表

| 事件 | 描述 |
| --- | --- |
| ionViewDidLoad | 当页面被加载后调用。这个事件仅在每个页面被创建时发生一次。如果离开一个页面但页面被缓存了，这个事件并不会在后面再次浏览时触发 |
| ionViewWillEnter | 当页面即将进入并成为激活状态时触发 |
| ionViewDidEnter | 当页面完全进入并已经激活时触发。无论这个页面是第一次加载还是一个缓存的页面都会触发 |
| ionViewWillLeave | 当将要离开页面、不再激活时触发 |
| ionViewDidLeave | 当完全离开页面、不再激活时触发 |
| ionViewWillUnload | 当页面即将被销毁、它的元素已经被移除时触发 |
| ionViewCanEnter | 在视图能够进入时触发。这个事件可以用于作为授权访问视图的“保护”措施，当这些视图能够进入之前我们需要进行权限检查 |
| ionViewCanLeave | 视图能够离开时触发。这个事件可以作为授权访问页面的“保护”措施，当这些视图能够离开之前我们需要进行权限检查 |

## 管道符

管道符，原先叫做“过滤器”，将一个值转换成新的值，比如将一个字符串进行本地化或将一个浮点数转换成现金格式：

```
<p>The author's birthday is {{ birthday | date }}</p>
```

如果 birthday 变量是标准的 JavaScript 日期对象，它可能是这个样子：Thu Apr 18 1968 00:00:00 GMT-0700 (PDT)。这当然不符合人类的阅读习惯。因此，在我们的模板的插值中，我们将 birthday 通过管道操作符（|）传递给日期管道函数，这样作者的生日就变成了 April 18,1968。

Angular 提供了一系列常用的管道操作，比如 DatePipe，UpperCasePipe，LowerCasePipe，CurrencyPipe 和 PercentPipe。它们在任何模板中都是可以直接使用的。

## @ViewChild

我们经常需要读、写子组件的值或者子组件的方法。当父组件 class 需要这些功能时，我们可以将子组件以一个 ViewChild 的形式注入父组件中：

```
import {Component, ViewChild} from '@angular/core';
import {UserProfile} from '../user-profile';

@Component({
  template: '<user-profile (click)="update()"></user-profile>',
  directives: [UserProfile]
})

export class MasterPage {
  // 传入我们想访问的组件
  // 在类中声明一个公共属性
  // 指定组件的类型
  @ViewChild(UserProfile) userProfile: UserProfile
  constructor() { }
  update(){
    this.userProfile.sendData();
  }
}
```

从 Angular Core 中导入 ViewChild 和 UserProfile 组件。在 @Component 修饰符中，我们必须设置 directives 属性为注入的组件的引用。我们的构造器中包含了 @ViewChild 修饰符，在这个修饰符中，我们将 userProfile 变量设置为注入的组件。

使用了这些代码之后，我们就能调用子组件的 sendData 方法了。

# 理解 ES6 和 TypeScript

近几年来，Web 开发者突然发现，有许多意图创建出“更好的”或者更能满足开发者需要的 JavaScript 版本出现了。CoffeeScript、AtScript、Dart、ES 6、TypeScript 等，都试图改进 JavaScript。这些语言都企图通过提供符合现代 App 开发的特性和功能来扩展 JavaScript。但每一种都必须要解决这个问题，即我们现在的浏览器所使用的 JavaScript 版本是所谓的 ECMAScript 5（ES 5），这样每一种语言都必须将它的代码转换成标准的 JavaScript。

为了能够使用上述这些现代语言，我们的代码必须转译成 ES 5 标准的 JavaScript。如果你从未听说过“转译”一词，那么它实际上是一个将某种编写好的语言转换成另一种语言的过程。

当然，如果你想使用下一代的 JavaScript，有两种最主要的选项供你选择：ES 6 或者 TypeScript。ES 6 是 JavaScript 的下一代的官方版本，它在 2015 年 1 月正式通过，随着时间的推移，它会逐渐成为浏览器支持的正式标准。另外一种语言是 TypeScript。TypeScript 是微软对 JavaScript 的扩展，它提供了强大的类型检查和面向对象特性。它的内核也使用了 ES 6。TypeSript 是 Angular 和 Ionic App 开发中的主要语言。

尽管我们当前的浏览器不支持这两种语言，但可以用一些工具比如 Babel 或 tsc 将我们的代码转译成标准 JavaScript。我们不需要关心这个过程，因为它已经内置在 Ionic 编译过程中了。

## 变量

在 ES6 中，我们定义变量的方式发生了改变。我们现在可以用 `let` 关键字指定一个变量。

在 ES5 中，变量只能用关键字 `var` 定义，变量的作用范围会扩散到最近的函数中。这经常会导致一些问题，因为变量可以在定义它的函数之外被访问。

```
for (var i = 0 ; i < 10; i++) {
    console.log(i);  // 输出 0-9
}
console.log(i); // 输出 10
```

变量 i 在循环结束之后仍然可以被访问，这会导致一些意外的发生。

通过 let 关键字，这个问题得到了解决，变量的作用域仅限于最近的块：

```
for (let i = 0 ; i < 10; i++) {
    console.log(i); //输出 0-9
}
console.log(i); // 未捕获的引用错误：i is not defined
```

现在，当循环完成后，i 不再对后面的代码有效。请尽可能使用 let 关键字来定义变量。

## 类

JavaScript 类从 ES6 开始引入，它是一个语法糖，基于 JavaScript 原有的 prototype。类语法并没有在 JavaScript 中引入新的面向对象继承机制。如果你曾经使用过其他面向对象语言比如 C# 或者 Java，新的类语法是非常类似的。请看例子：

```
class Rocket {
  landing(location) {
  }
}

class Falcon extends Rocket {
  constructor() {
    super();
    this.manufacturer = 'SpaceX';
    this.stages = 2;
  }
  landing(location) {
    if ((location == 'OCISLY') || (location == 'JRTI')){
      return 'On a barge';
    } else {
      return 'On land';
    }
  }
}

class Antares extends Rocket {
  constructor() {
    super();
      this.manufacturer = 'OrbitalATK';
      this.stages = 2;
    }
    landing(location) {
      console.log('In the ocean');
```

```
    }
}
```

## Promise

`Promise` 对象用于延迟操作和异步操作。一个 Promise 表示一种未完成的操作，这个操作有可能在未来完成。当需要和远程服务器打交道或者加载本地数据时，Promise 非常有用。它提供了一种比传统回调方法更简单的方式来处理异步操作。

一个 Promise 可能有 3 种状态：

*Pending*

Promise 的结果还不确定，因为异步操作还未返回结果。

*Fulfilled*

异步操作已经完成，Promise 已经有确定值。

*Rejected*

异步操作失败，Promise 永远不会实现（Fulfilled）。在 rejected 状态时，Promise 会用一个 reason 来表示操作为什么失败。

Promise 的主要 API 是它的 `then` 方法，这个方法注册了一个回调用于接收结果值或者 Promise 不能被满足（fulfilled）的原因。

假设我们有一个 `sayHello` 函数是异步的，需要从 Web 服务器基于用户当前地理坐标查找当前问候语，它就可以返回一个 Promise：

```
var greetingPromise = sayHello();
greetingPromise.then(function (greeting) {
    console.log(greeting);    // 'Hello in the United States'
});
```

这种方法的好处是，在函数等待服务器响应的时候，后面的代码仍然可以运行。

如果发生错误，比如网络断开，无法从服务器获取问候语，你可以注册一个失败回调，使用 Promise 的 `then` 方法的第二个参数：

```
var greetingPromise = sayHello();
greetingPromise.then(function (greeting) {
    console.log(greeting);     // 'Hello in the United States'
}, function (error) {
```

```
    console.error('uh oh: ', error); // '该死!'
});
```

如果 sayHello 成功，问候语将输出到控制台，如果失败，错误原因（即 error）会通过 console.error 方法输出。

## Observable

Angular 有许多服务使用 Observable 而不使用 Promise。Observable 是用 RxJS 库实现的。和 Promise 用于解决单个值的同步不同，一个 Observable 用于解决多个值的同步（实时）。

此外，Observable 可以取消，能够用 API 中的重试操作比如 retry 和 retryWhen 进行重试。而 Promise 需要调用者调用返回 Promise 的那个函数才能提供 retry 功能。

## 模板字符串

Angular 的一个特点是它内置了模板引擎。许多时候，模板是以外部文件的方式存在的。但有时候，将模板直接写在代码行中可能更直观。这样在书写大段行内模板时就可以不使用字符串连接符和对单双引号进行转义了。

ES6 支持在字符串头尾加上反引号：

```
let template = `
  <div>
    <h2>{{book.name}}</h2>
    <p>
      {{book.summary}}
    </p>
</div>
`;
```

模板字符串不一定是静态的。你可以通过 $( 表达式 ) 的方式进行字符串插入：

```
let user = {name:'Rey'};
let template = `
  <div>Hello, <span>${ user.name }</span></div>
`;
```

**模板表达式：ES6 与 Angular**

ES6 的模板表达式仅仅是字符串替换。如果你想计算一个函数或者测试某个条件，请使用 Angular 模板表达式。

## 箭头函数

箭头函数使我们的代码更简洁，简化函数作用域和 this 关键字。通过箭头函数，我们可以不用输入 function 关键字、return 关键字（显而易见的）以及大括号。

在 ES5，我们写一个这样的乘法函数：

```
var multiply = function(x, y) {
  return x * y;
};
```

但在 ES6，使用新的箭头函数语法，我们可以这样写一个类似的函数：

```
var multiply = (x, y) => { return x * y };
```

这个箭头函数的例子允许我们用更少的代码、接近一半的输入量实现同样的效果。

箭头函数的常用场景之一，就是数组操作。比如这个简单的数组对象：

```
var missions = [
  { name:'Mercury', flights:6 },
  { name:'Gemini', flights:10 },
  { name:'Apollo', flights:11 },
  { name:'ASTP', flights:1 },
  { name:'Skylab', flights:3 },
  { name:'Shuttle', flights:135 },
  { name:'Orion', flights: 0 }
];
```

在 ES5 中，我们可以这样，只使用对象中的航班名称或航班号创建一个新的数组：

```
// ES5
console.log(missions.map(
  function(mission) {
    return mission.flights;
  }
)); // [6, 10, 11, 1, 3, 135, 0]
```

如果用箭头函数来写，我们的代码可以更加简洁和易读：

```
// ES6
console.log(missions.map(
  mission=>mission.flights
)); // [6, 10, 11, 1, 3, 135, 0]
```

## 类型

TypeScript 是一个类型化的语言，允许你在编译时进行类型检查。默认，TypeScript 支持 JavaScript 的原生类型：string、number 和 boolean。

```
let num: number;
let str: string;
let bool: boolean;

num = 123;
num = 123.456;
num = '123'; // 错误

str = '123';
str = 123; // 错误

bool = true;
bool = false;
bool = 'false'; // 错误
```

TypeScript 也支持类型化数组。语法是在有效的类型声明之后添加 [] 后缀（比如 boolean[]）：

```
let booleanArray: boolean[];

booleanArray = [true, false];
console.log(booleanArray[0]); // true
console.log(booleanArray.length); // 2
booleanArray[1] = true;
booleanArray = [false, false];

booleanArray[0] = 'false'; // 错误！
booleanArray = 'false'; // 错误！
booleanArray = [true, 'false']; // 错误！
```

## 特殊类型

除了原始类型以外，在 TypeScript 中还有几个类型有着特殊的含义。它们是 any、null、undefined 和 void。

```
let someVar: any;
```

```
// 可以用 any 保存任意类型
someVar = '123';
someVar = 123;
```

JavaScript 字面量 null 和 undefined 可以看成是一种 any 类型：

```
var str: string;
var num: number;

// 这些字面量可以赋给任意类型的变量
str = undefined;
num = null;
```

# 类型化函数

不仅仅变量可以类型化，函数返回值也可以类型化：

```
function sayHello(theName: string): string {
    return 'Hello, '+theName;
}
```

如果你调用 sayHello 方法并将结果赋给一个类型不一致的变量，编译器会抛出一个错误。

## :void

:void 用于表示函数没有返回类型：

```
function log(message): void {
    console.log(message);
}
```

# 小结

无论 Angular 还是 TypeScript，它们包含的内容实在是太多了，本章只是一个简单的介绍而已。很多时候，为了开发一个功能完整的 App，你需要用到这两种技术的资源会有很多。让我们来回顾一下本章都介绍了哪些知识点。

我们介绍了 Angular 的新的组件模型，它的模板、新的数据绑定方法以及组件的生命周期。此外，我们也介绍了一些 ES6 和 TypeScript 的新特性，包括类、Promise、箭头函数以及类型化的变量和函数。

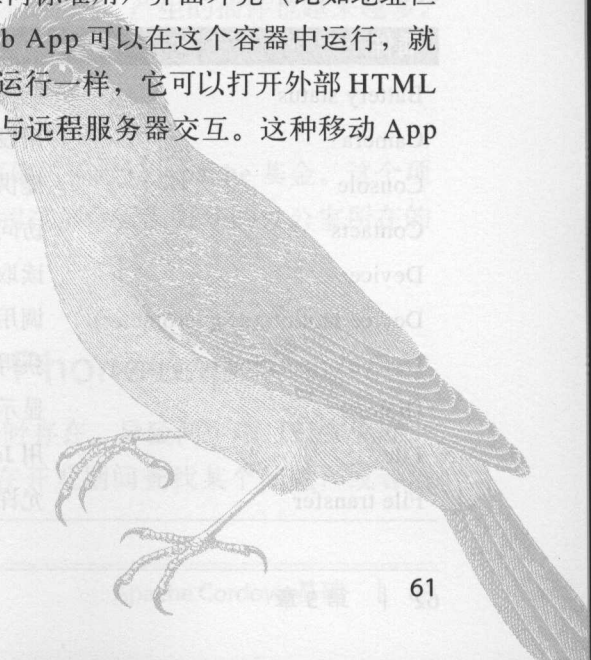
第 5 章

# Apache Cordova 基础

Ionic 框架构建于另外两种技术之上：Angular 和 Apache Cordova。在本章，我们将学习 Apache Cordova 是什么以及它如何与 Ionic 交互。

Apache Cordova 是一个开源框架，允许移动 App 开发者用 HTML、CSS 和 JavaScript 内容创建针对各种移动设备的本地应用。让我们来看看它的机制是什么。

Cordova 会将你的 Web 应用渲染到原生 Web view 中。Web view 是一个原生 App 组件（和一颗按钮或一个 tab bar 类似），该组件用于在原生 App 中显示 Web 内容。你可以把 Web view 看成是没有任何标准用户界面外壳（比如地址栏和状态栏）的 Web 浏览器（见图 5-1）。Web App 可以在这个容器中运行，就像任何其他类型的 Web 应用在移动浏览器中运行一样，它可以打开外部 HTML 页、执行 JavaScript 代码、播放媒体文件、与远程服务器交互。这种移动 App 类型通常被称作混合 App。



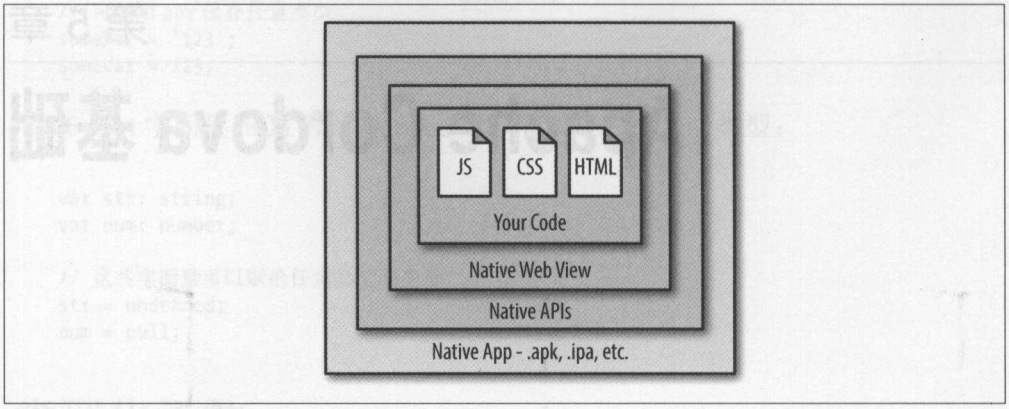



图 5-1：Cordova App 实际上完全是一个原生 App

通常，基于 Web 的 App 是在沙盒中运行的，也就是说它们无法直接访问设备上的各种硬件和软件功能，比如移动设备上的联系人数据库。这个数据库上的人名、电话号码、邮箱和其他信息是无法被 Web App 所访问的。除了提供基本框架允许 Web App 运行在一个原生 App 中之外，Cordova 还提供了允许访问各种设备特性比如联系人数据库的 JavaScript API。这些能力通过一系列插件来暴露给开发者。插件提供了一个 Web App 和设备原生能力之间的桥接层。在本书后面的内容中，我们会学习它们的使用。Cordova 项目维护了一个核心的插件集合，但更多的功能是通过第三方插件来提供的（例如，NFC 通信、压感触控、推送通知），见表 5-1。

表 5-1：核心插件

| 插件 | 功能 |
| --- | --- |
| Battery status | 监控设备电池状态 |
| Camera | 用设备相机拍照 |
| Console | 提供高级控制台日志 |
| Contacts | 访问设备的联系人数据库 |
| Device | 读取设备相关信息 |
| Device motion (accelerometer) | 调用设备的运动传感器 |
| Device orientation (compass) | 获得设备指向的方向 |
| Dialogs | 显示设备通知 |
| File | 用 JavaScript 访问原生文件系统 |
| File transfer | 允许 App 上传、下载文件 |

表 5-1：核心插件（续）

| 插件 | 功能 |
| --- | --- |
| Geolocation | 允许 App 感知位置变化 |
| Globalization | 允许对象的呈现内容和区域位置相关 |
| InAppBrowser | 用另一个应用内浏览器实例来访问 URL |
| Media | 录音、回放音频文件 |
| Media-capture | 通过设备的媒体捕获程序录制媒体文件 |
| Network connection | 快速检测网络状态和蜂窝网络信息 |
| SplashScreen | 显示、隐藏 App 的启动界面 |
| StatusBar | 显示、隐藏、配置状态栏背景的 API |
| Vibration | 振动设备的 API |
| Whitelist | 实现一个允许 App 的 Web view 能够跳转的白名单策略 |

## Cordova（即 PhoneGap）历史

开发者们经常将 Apache Cordova 和 PhoneGap 二者混淆。为了澄清这种混淆，我们需要了解这个项目的起源。在 2008 年年末，Nitobi 的几个工程师参加了一个位于旧金山 Adobe 公司的 iPhone 开发训练营。他们产生了一种用原生 Web view 作为外壳以便在原生环境中运行 Web App 的念头，这个试验成功了。经过几个月后，他们扩大了他们的研究成果并利用这种方法创建了一个框架。他们把这个项目称为 PhoneGap，因为它能够让 Web 开发者将 Web App 和设备的原生特性之间的鸿沟桥接起来。随着这个项目的不断成熟，产生的插件也越来越多，允许访问的手机功能也越来越多。其他贡献者不断添砖加瓦，增加了它所支持的移动平台。

2011 年，Adobe 收购了 Nitobi，PhoneGap 框架被捐献给 Apache 基金。这个项目最终更名为 Cordova（这其实是 Nitobi 公司在加拿大温哥华的办公室所在的街道名）。

## Apache Cordova 与 Adobe PhoneGap

因为 Apache Cordova 和 Adobe PhoneGap 同时存在，导致两个项目经常搞混。因为这两个项目是如此的纠结不清，当我们在开发期间查找某个问题，或者需

要在关键字中添加 Cordova 和 PhoneGap 才能搜索到答案，或者在阅读某个文档时，经常被这两个名字弄晕。

要理解这两者的区别，可以参考苹果的 Safari 浏览器和它的内核——开源的 WebKit 引擎之间的关系。这两个情况真的很像：Cordova 是开源的，PhoneGap 是 Adobe 注册的。最终，二者间只有轻微不同。在命令行界面上二者有所不同，但功能相同。唯一需要注意的是，你不能在同一个项目中混用这两个框架。虽然不是像电影大片中穿过河流那么危险，但在同一项目中同时使用 Cordova 和 PhoneGap 只会带来麻烦。

### PhoneGap 还是 Cordova?

我们建议使用 PhoneGap 作为第一搜索关键字，因为在查找问题时，它就是 Cordova 原本的名称。

两个项目的最主要区别是 Adobe PhoneGap 有一些付费服务，最明显的就是 PhoneGap Build 服务。这是一个托管服务，允许你远程将 App 编译成本地二进制，这样就不需要在本地安装每种移动平台的 SDK 了。PhoneGap CLI 可以调用这个服务，而 Cordova CLI 不能。

另外的区别是 PhoneGap CLI 可以和 PhoneGap developer app（*http://app.phonegap.com/*）一起使用。这个免费的移动 App 允许你的 App 在不需要编译的情况下直接运行到设备上。这提供了一种非常简单的在设备上调试和测试 App 的方法。不过也别担心，Ionic 也提供了类似的功能（*http://view.ionic.io/*）给我们使用，可以在开发中使用它。

在这本书中，我们将使用 Cordova 命令行工具而不使用 Ionic 命令行工具。部分原因就是由于 Ionic CLI 基于 Cordova CLI，而不是 PhoneGap CLI。

## 深入了解 Cordova

曾经，要配置一个 Cordova 项目是非常困难的。首先需要在每个平台的 IDE 中创建一个本地 App 项目，然后在此基础上修改以便支持 Cordova 接口。随着命令行工具的出现，这个过程被简化了。CLI 会搭建一个基本的项目脚手架并将

它配置成为支持你指定的所有移动平台。Cordova CLI 还允许我们轻松地集成和管理项目中的插件。最后，CLI 允许我们快速地编译 App 并在模拟器或真机上运行。我们会在后面通过示例项目学习这些命令。

## 配置你的 Cordova App

每个 Cordova App 都是通过 *config.xml* 文件来配置的。这个全局的配置文件控制了 App 的方方面面，从 App 的图标、插件到和平台相关的各种设置。它是基于 W3C 的 Package Web Apps(Widgets) 规范（*http://www.w3.org/TR/widgets/*）的。Ionic CLI 在搭建脚手架的过程中会生成一个基本的 *config.xml* 文件。

在开发你的 Ionic App 时，你可能需要修改这个文件，添加新的元素。一个常见的例子是修改 App 名称、App ID 字符串或者添加平台相关的设置，比如 `android-minSdkVersion`。

关于 *config.xml* 的更详细的介绍请参考附录 A。

## 设备的可访问性（即插件）

Cordova 的真正强大之处是它的插件库。在编写本书的时候，大概有超过 1250 个插件。但是，什么是 Cordova 插件？这里给出了 Cordova 官方网站上的定义：

> 一个插件是一些额外的代码，允许 JavaScript 和原生组件交互。它们允许你的 App 能够调用单纯的 Web App 所不具备的原生设备的能力。

在前面我们提到过，有两套插件：核心插件和第三方插件。这里又要说一段历史了。直到 PhoneGap 3.0（Cordova 预览版），代码库中包含有使用 Web view 和原生 App 进行通信的代码，以及一些“主要的”的设备插件。从 PhoneGap 3.0 之后，所有的插件被抽离成单独的元素。这种改变意味着所有的插件都可以按需升级，而无需更新整个代码库。这些初始插件被称作“核心”插件。

但是，大约 1220 个插件，它们都可以用来做什么呢？它们的范围十分广泛：包括了蓝牙连接、推送通知、TouchID 和 3D Touch 等。

## 界面组件：缺失的拼图

虽然 Cordova 为开发者提供了大量的功能比如跨平台、对 Web 进行扩展、调用设备原生功能，但它仍然缺少一个重要的组件：用户界面组件。默认只提供基本的 HTML 控件，Cordova 不提供任何可以在原生 SDK 中找到的任何组件。如果你想拥有一个 tab bar 组件，你要么编写 HTML 标签、CSS 样式及其负责管理这些视图的 JavaScript，要么就使用第三方框架，比如 Ionic。

UI 层的缺失往往是使用 Cordova 进行开发中的最困难的工作。随着 Ionic 框架的出现，开发者拥有了可在编写自己的移动 App 中使用的一流界面工具包。在一个 Cordova 项目中可以使用别的框架，但本书只介绍 Ionic。如果你想看看其他解决方案，你可以看看 OnsenUI、Framework7、ReactJS。对于我来说，我喜欢用的是 Ionic 框架及其服务。

## 为什么不使用 Cordova

尽管 Cordova 是一个非常强大的移动 App 构建方案，但它不能适用任何情况。为了让你的 App 表现卓越，理解这个框架的优缺点非常重要。Cordova 应用离真正的原生层还隔着几层。因此，要开发一个高性能的 Cordova App 必须格外小心。在 PhoneGap 的早些时候这种说法无疑是正确的，当时的设备性能还不够强大。现在的移动设备可以使用的计算能力已经强太多了。

也就是说，你还不能用 Cordova 开发出类似“愤怒的小鸟”，或者是“逃离神庙”那样的 App，但已经可以用 Cordova 开发出类似 Quiz Up、Untappd 或者 Trivia Crack 那样的 App 了。

## 理解 Web 标准

另一件事情是 WebView 到底能做到哪一步？尽管大部分移动 WebView 都紧跟已知的 Web 标准，但出现某些问题或不正常也是可以理解的。如果你曾经做过某种现代 Web 开发，你可能知道 *http://caniuse.com*。

这是一个检查某个 HTML5 特性是否在浏览器中可用，或者浏览器是否支持某个 CSS 属性的网站。例如，如果我们想在 App 中使用可伸缩的矢量图形（SVG）

以解决大量屏幕尺寸和分辨率适配的问题，可以浏览这个网址，并查看每个移动浏览器从什么时候开始支持它（见图 5-2）。

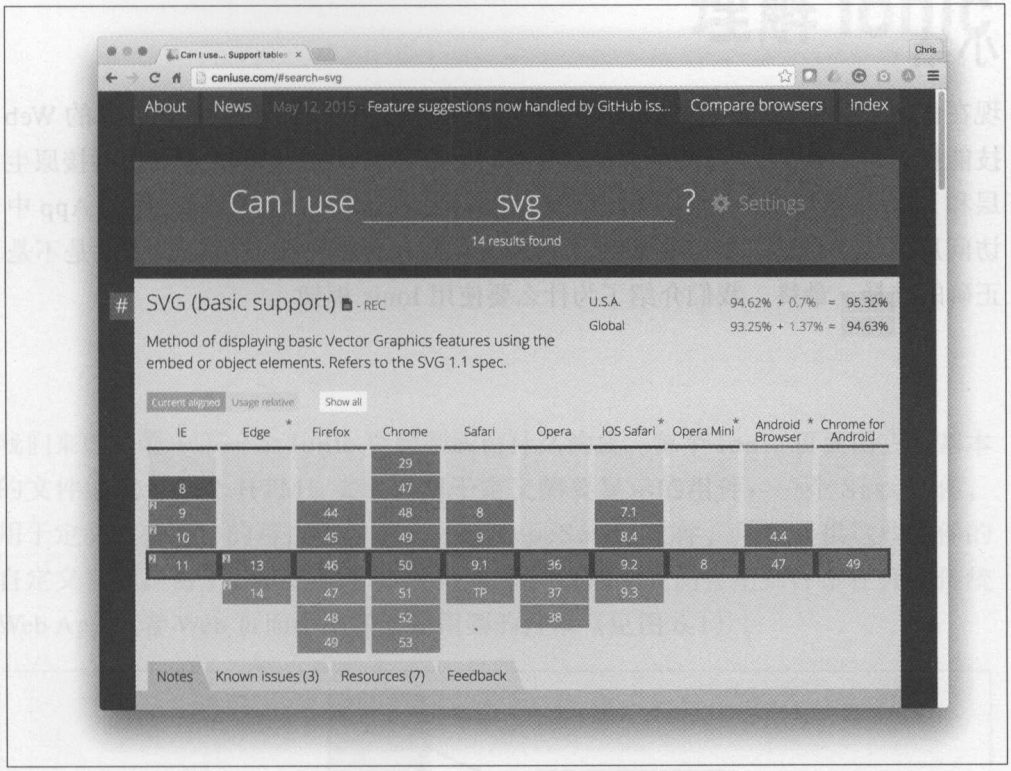



图 5-2：各种浏览器对 SVG 的支持

这样，我们就会知道 iOS 已经从好几个版本以来就支持 SVG 了，而 Android 只有在最近才开始支持。

需要注意的是，Cordova 所用的 WebView 到底来自哪里。当你编译 App 时，Cordova 并不会在你的 App 中放进一个 WebView，它其实使用的是设备上内置的 WebView。也就是说，当前的 iOS 设备会使用它自己的 WebKit 版本，而 Android 设备会使用 Chromium，其他设备也可能使用 WebKit。

Crosswalk 项目（*https://crosswalk-project.org/*）目前已经停止，但如果你要在老的 Android 设备上开发 App 的话，这个项目仍然是一个很好的解决方案。这个插件会将 Cordova 项目中的默认的 WebView 替换成最新版本的 Google Chromium，从而获得最新的浏览器特性。尽管这个方法会明显增加 App 的尺寸，

却有助于规范你能够调用的 Web 特性。如果你的 App 会在较老的 Android 设备上运行，Crosswalk 是一个很好的解决浏览器兼容性的办法。

## 小结

现在你已经对 Apache Cordova 有了一个很好的了解，也学习了如何将你的 Web 技能投入移动 App 的开发中。在这一章，我们学习了 Cordova 是如何桥接原生层和 Web 技术层的。我们还讨论了 Cordova 插件，插件允许我们在混合 App 中访问原生设备能力。我们还讨论了这种方案对于你的开发需求来说到底是不是正确的选择。最终，我们介绍了为什么要使用 Ionic 框架。

第 6 章

# 理解 Ionic

我们来好好看一下一个 Ionic 页面到底由什么构成。每个 Ionic 页面由三种基本的文件组成：一个 HTML 文件，用于定义需要显示的组件；一个 Sass 文件，用于定义这些组件的可视化样式；一个 TypeScript 文件，用于提供这些组件的自定义功能。由于 Ionic 是基于 Web 技术构建的，我们会用到许多在开发传统 Web App 或者 Web 页面中同样会使用到的技术（见图 6-1）。

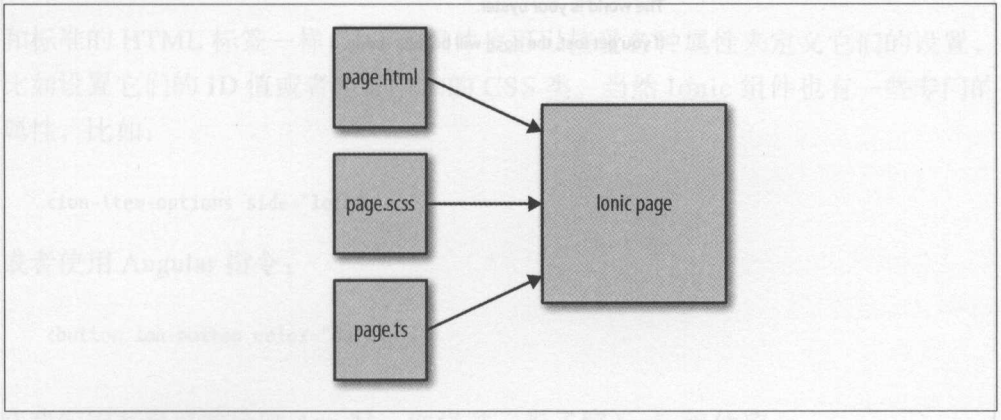



图 6-1：基础 Ionic 页面结构

## HTML 的构成

和传统的 HTML 文件不同，不需要使用 `<head>` 标签和其他类似导入 CSS 文件或代码库的元素，也不需要使用 `<body>` 标签、`<!DOCTYPE html>` 或 `<html lang="en" dir="ltr">` 标签。

这个 HTML 文件是在 App 容器中渲染的，因此我们不需要它。我们只需要定义真正需要展示给用户的组件。这些组件混合了传统 HTML 标签和自定义标签，这些自定义标签用于定义 Ionic 组件。这是一个 Ionic 页面的例子：

```
<ion-header>
  <ion-navbar>
    <ion-title>
      Ionic Blank
    </ion-title>
  </ion-navbar>
</ion-header>

<ion-content padding>
  The world is your oyster.
  <p>
    If you get lost, the <a href="http://ionicframework.com/docs/v2">↵
    docs</a> will be your guide.
  </p>
</ion-content>
```

渲染后的页面显示如图 6-2 所示。

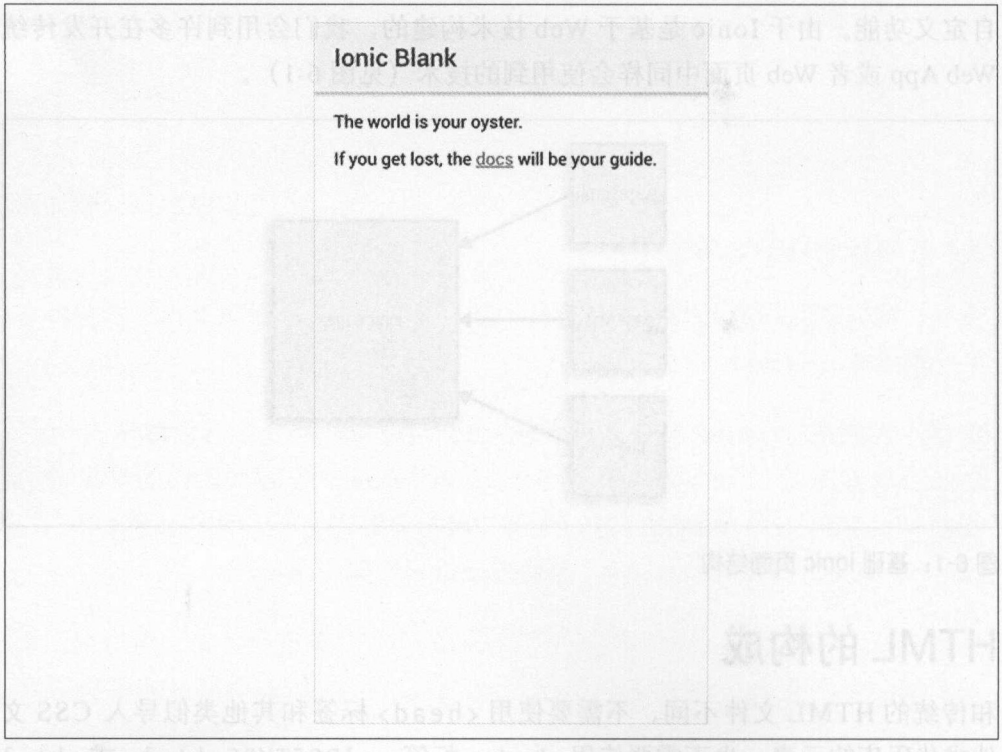



图 6-2：渲染后的页面

你可以看到它是由标准 HTML（<p> 和 <a>）和 Ionic 标签（<ion-header>、<ion-content> 等）混合而成。

## Ionic 组件

Angular 的一个功能是能够通过自定义标签来扩展 HTML 标签。Ionic 框架利用这个功能创建了一整套移动组件。这些组件包括 <ion-card>、<ion-item-sliding> 和 <ion-segment-button>。关于 Ionic 组件库的完整介绍，请看附录 C，以及 Ionic 文档（*http://ionicframework.com/docs/*）。所有的 Ionic 组件都带有前缀 ion-，这样要搜索它们会非常简单。

**Ionic 标签自动完成**

大部分代码编辑器都可以从某种程度上扩展自动完成或代码提示功能。通常需要下载编辑器的插件才能开启 Ionic 框架的代码提示功能。

此外，每个组件都会在运行时根据它们自定义的 CSS 样式和功能解析成标准的 HTML 标签。

和标准的 HTML 标签一样，Ionic 组件也可以接受各种属性来定义它们的设置，比如设置它们的 ID 值或者定义附加的 CSS 类。当然 Ionic 组件也有一些专门的属性，比如：

```
<ion-item-options side="left">
```

或者使用 Angular 指令：

```
<button ion-button color="dark">
```

在我们编写自己的示例 App 时，你将进一步了解 Ionic 组件库。

## 理解 SCSS 文件

一个 Ionic App 的界面用 CSS 进行定义。但是，这个 CSS 实际是用 Sass 或 Syntactically Awesome Style Sheets 来创建的。如果你从来没用过 Sass，那么它比起 CSS 来有几个好处。这就包括了变量声明，比如 $company-

brand:#ff11dd。然后就可以引用这个变量而不用直接使用颜色值。如果我们需要修改这个颜色，我们可以只在一个地方修改它而无需在多个文件中修改。

### CSS 变量命名

如果经常使用类似 $company-red 这样的名字来命名变量，你可以想象一下，当品牌颜色突然需要从红色改变为绿色时会有多尴尬？

所有的 Ionic 组件都使用 Sass 变量来设置样式。例如，我们可以在 *app.variables.scss* 文件中修改 $list-background-color 的值。Ionic 团队做了大量工作，确保每个 Ionic 组件的样式都易于设置。要查看完整的可用的 Sass 变量列表，请参考 Ionic 文档。

Sass 支持一种改良语法，用于编写嵌套的 CSS。例如：

```
nav {
  ul {
    margin: 0;
    padding: 0;
    list-style: none;
  }

  li { display: inline-block; }

  a{

    display: block;
    padding: 6px 12px;
    text-decoration: none;
  }
}
```

转换之后会变成：

```
nav ul {
  margin: 0;
  padding: 0;
  list-style: none;
}

nav li {
  display: inline-block;
}
```

```
nav a {
  display: block;
  padding: 6px 12px;
  text-decoration: none;
}
```

用这种方式书写 CSS 更加简洁，更节省时间。

通常，页面的 .scss 文件用于定义和某个页面相关的 CSS，例如，当你需要一个和登录页面不同样式的按钮的时候，可以在页面相关的 *.scss* 文件中进行定义，并将相关的元素通过一种更符合逻辑的方式组织在一起。对于那些 App 级别的主题样式，则可以在 *app.core.scss* 文件中进行样式的定义。

我们会在构建自己的示例 App 时更深入地讨论关于自定义样式的内容。

## 理解 TypeScript

编写 Ionic 页面的最后一个元素是相关的 TypeScript（.ts）文件。这个文件用于编写和页面交互逻辑相关的 Angular/TypeScript 代码。

这个文件会定义所有需要导入到页面中的代码模块，一般包括那些需要在代码中用到的组件（比如导航到新的页面）或者负责提供必要功能的 Angular 模块（比如发送一个 HTTP 请求），也可以定义屏幕上会用到的基本组件。如果我们需要和用户输入打交道，那么代码中还需要添加对应的文件。这是一个基本的 Ionic 的 *.ts* 文件例子：

```
import { Component } from '@angular/core';
import { NavController } from 'ionic-angular';

@Component({
  selector: 'page-home',
  templateUrl: 'home.html'
})

export class HomePage {

  constructor(public navCtrl: NavController) {

  }

}
```

这些 TypeScript 代码仅仅定义了一个 HomePage 组件，并将它和模板进行绑定。

在我们的示例 App 中，将大量使用到 TypeScript 代码，因此这里就不过多介绍这个元素了。

## 小结

通过本章，你应该对构成 Ionic 页面的三个关键元素：HTML 文件、Sass 文件和 .ts 文件有一个较好的了解了。接下来，让我们开始编写我们真正意义上的第一个 Ionic App 吧！

# 第 7 章

# 编写我们的 Ionic2Do App

我们的开发环境已经配置好，对 Angular 2 和 Apache Cordova 也有了一个基本的了解，终于可以开始创建我们的第一个 Ionic 2 App 了。为了不破坏一贯的传统，我们将编写一个经典的 to-do 列表管理 App。你可能奇怪为什么要用这种随处可见的例子来编写示例 App？其中一个原因就来自于你，因为编写熟悉的 App 能够让你将 Ionic 和曾经用过的其他语言或框架进行一个很好的比较。另外一个原因则是与在页面上显示一个简单“Hello Wolrd”比起来，to-do App 足够复杂。

首先，我们需要创建一个新的 Ionic 项目，从一个空白模板开始创建：

```
$ ionic start Ionic2Do blank --v2
```

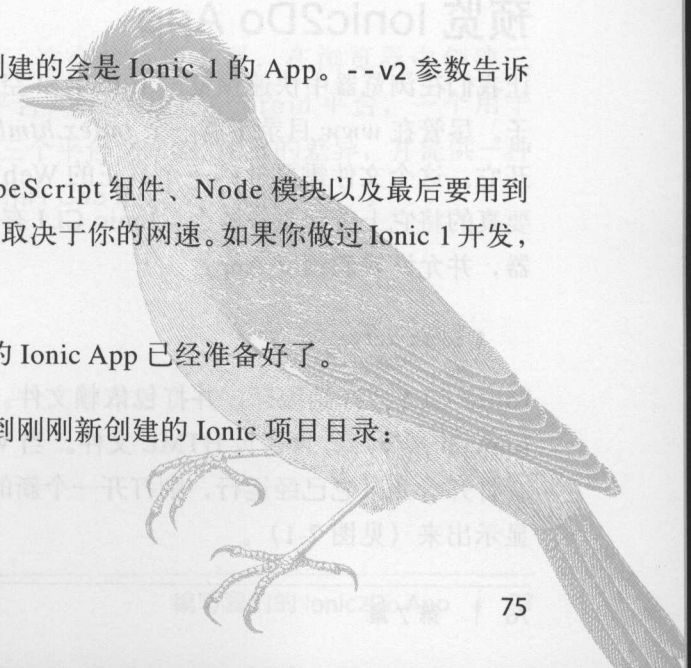
确认命令中包含 `--v2` 参数；否则你创建的会是 Ionic 1 的 App。`--v2` 参数告诉 Ionic CLI 我们要创建 Ionic 2 项目。

CLI 会开始下载项目的各种内容：TypeScript 组件、Node 模块以及最后要用到的 Cordova 组件。这个过程需要的时间取决于你的网速。如果你做过 Ionic 1 开发，你可能会觉得这个过程比以前更长。

当所有包下载完，CLI 会告诉你，你的 Ionic App 已经准备好了。

然后，你需要将当前的工作目录切换到刚刚新创建的 Ionic 项目目录：

```
$ cd Ionic2Do
```



我不止一次忘记过这个简单步骤，甚至还奇怪为什么接下来的Ionic命令会出错。当它完成时，CLI也会有一个友好提示。

## 添加平台

如果你使用Mac进行开发，Ionic CLI会自动包含iOS平台。因为在Windows上不可能编译iOS平台，所以CLI不可能在Windows下添加iOS平台。而Android既能在Mac上也能在PC上编译，可以在我们的项目中添加Android平台：

```
$ ionic platform add android
```

这会在项目中添加一个platforms目录和一个*platforms.json*文件。Platforms目录为每个平台创建一个子目录。这时你可以打开*platforms>android*目录，你会看到在Android OS下创建的我们的App所要用到的所有平台相关内容：Gradle文件、*AndroidManifest.xml*等。

要添加iOS平台，我们只需要修改CLI命令中的平台名称：

```
$ ionic platform add ios
```

现在，platforms目录中会包含一个iOS目录，这个目录中包含了和iOS平台相关的文件，比如Xcode项目文件。

## 预览 Ionic2Do App

让我们在浏览器中快速预览一下Ionic 2空白模板创建出的App大概是个什么样子。尽管在*www*目录下有一个*index.html*文件，但我们不能直接在浏览器中打开它。这个文件需要放到一个真正的Web服务器上才能打开。幸好，我们不需要真的将它上传到服务器上，Ionic CLI有一个命令会启动一个本地的Web服务器，并允许我们预览App：

```
$ ionic serve
```

Ionic CLI会开始编译，并打包依赖文件。然后将Sass文件编译成CSS，插入IonIcon字体库，再拷贝HTML文件。当Web服务器启动好后，Google Chrome会打开（如果它已经运行，则打开一个新的标签页），然后我们的Ionic App会显示出来（见图7-1）。

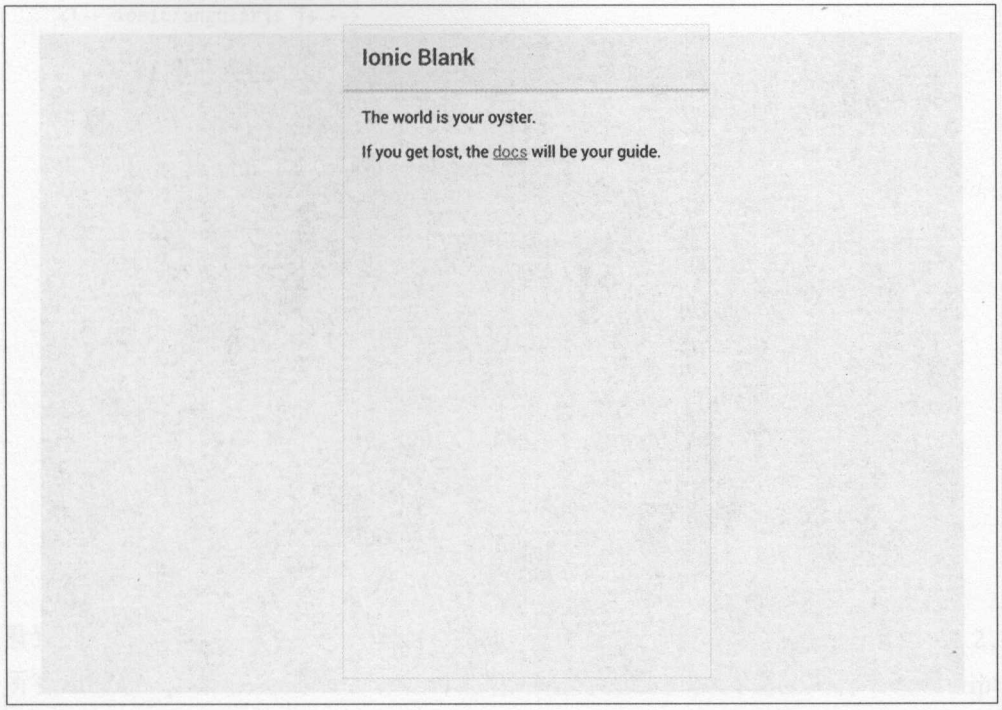



图 7-1：Ionic 的空白模板

服务器会监听你对项目文件的改动，并在你保存文件时重新加载 App。

此外，你可以在命令最后添加一个 `--c` 参数，这会启动控制台日志，或者用 `--s` 参数打开服务器日志。

另外我还经常使用一个 `--lab` 参数。这会告诉服务器，在浏览器中创建三个 Ionic App 拷贝：一个用于 iOS 平台，一个用于 Android 平台，一个用于 Windows 平台。这会允许你快速预览三个平台之间的 UI 上的差异，并提供一种测试手段，允许你测试你正在项目使用的 CSS 的改变（见图 7-2）。

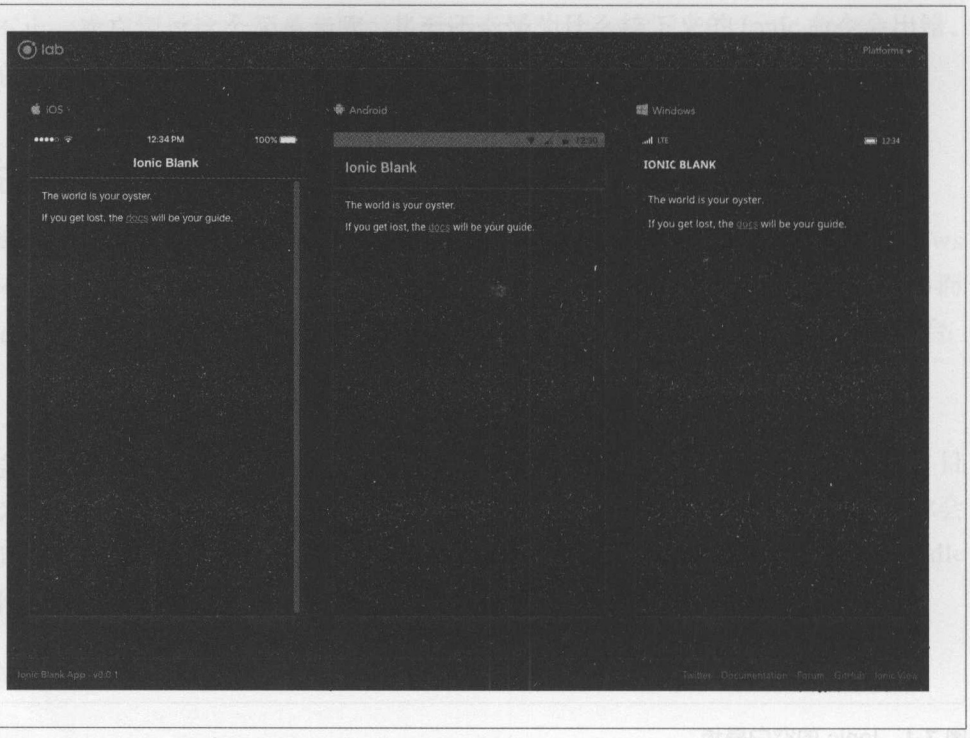



图 7-2：在 Ionic serve 命令中使用 --lab 参数

要退出服务器，在终端窗口中使用 **Control-C** 组合键。

### 预览某个平台

在使用 `$ ionic serve` 命令时，你的 App 会使用 Android 作为指定的平台。要在不同的平台下看到 App 渲染的结果，可以在命令行末尾加上 `--platform=platformname`。值可以是 `ios`、`android` 和 `windows`。

让我们回到项目目录，看看其他文件以及它们之间的关系。

## 理解 index.html 文件

首先来看一下 *www/index.html* 文件。在 Ionic 1 中，在这个文件中会放许多东西。下面是一个 Ionic 1 App 中的 script 标签的一部分：

```
 <!-- ionic/angularjs js -->
<script src="lib/ionic/js/ionic.bundle.js"></script>
<script src="lib/ionic-service-core/ionic-core.js"></script>
<script src="lib/ionic-service-analytics/ionic-analytics.js"></script>
 <!-- ngCordova -->
<script src="lib/ngCordova/dist/ng-cordova.min.js"></script>
<!-- cordova script (this will be a 404 during development) -->
<script src="cordova.js"></script>
<!-- App js -->
<script src="js/app.js"></script>
<!--Factories/Services-->
<script src="js/hikedata.js"></script>
<script src="js/mapdata.js"></script>
<script src="js/appStatus.js"></script>
<script src="js/geoLocService.js"></script>
<!--Controllers-->
<script src="views/home/home.js"></script>
<script src="views/hikes/hike.js"></script>
<script src="views/hikelist/hikelist.js"></script>
<script src="views/hikedetails/hikedetails.js"></script>
<script src="views/map/map.js"></script>
<script src="views/about/about.js"></script>
```

看到了吧，在开发中所有的 script 标签都被加在了 *index.html* 文件中。在 Ionic 2，所有 JavaScript 加载都由 CLI 和编译进程来进行控制。我们现在只有 3 个 script 标签，毫无疑问这个列表更好管理。事实上，我们不需要再去添加每个 Angular 元素的 script 标签，编译进程会生成一个单独的 *main.js* 文件，将所有代码集合在一起：

```
<!-- cordova.js required for cordova apps -->
<script src="cordova.js"></script>

<!-- The polyfills js is generated during the build process -->
<script src="build/polyfills.js"></script>

<!-- The bundle js is generated during the build process -->
<script src="build/main.js"></script>
```

如果我们去查看 `<body>` 标签中的内容，只会看到一个标签，`<ion-app>`。虽然只用了一个组件，我们整个 App 都会被加载。

从 Ionic 1 到 Ionic 2 的一个改变是，CSS 默认是用 Sass 文件生成的。在 Ionic 1 中，你必须开启这个过程。现在在 Ionic 2 中，Sass 中的每样东西都被编译成一个 CSS 文件：

```
<link href="build/main.css" rel="stylesheet">
```

从组件化开发的观点看来，几乎所有的代码都放在了 App 组件中。我们来进一步看下新的 *src* 目录。

在 *src* 目录中，我们会看到 4 个目录：*app*、*assets*、*pages* 和 *theme*，还有 4 个文件：*declarations.d.ts*、*index.html*、*mainifest.json* 和 *service-worker.js*。

## 探索 App 目录

在 Ionic 2 beta 版中，有一个 *app.ts* 文件，包含了 App 的通用 Angular 引导代码。转换到 NgModule 之后，Angular 固化了它的目录结构，Ionic 目录结构随之改变。现在，App 初始化文件被放到了 app 目录，App 的剩余部分被放到 *pages* 目录。

在 app 目录中，我们会看到 5 个文件。表 7-1 对它们分别进行了一个简单介绍。

表 7-1：文件概览

| | |
|---|---|
| app.component.ts | 这个文件包含了 App 初始化准备使用时的基本组件 |
| app.html | 这个文件定义了初始的 App HTML 标签 |
| app.module.ts | 这个文件定义了初始的模块、提供者和入口组件 |
| app.scss | 这个文件定义了全局 CSS 样式 |
| main.ts | 这个 TypeScript 文件在开发期间使用，用于加载 App |

让我们详细看一下这些文件，看看它们到底是干什么的。首先，用编辑器打开 *app.module.ts*。

### Visual Studio Code

开发者与代码编辑器之间的关系尤其重要。在使用 Ionic 2 时，我喜欢用微软的 Visual Studio Code 作为编辑器。它是免费的，可以从 *https://code.visualstudio.com/* 下载。毫无疑问，它支持 TypeScript、Cordova 和 Ionic。

这个文件定义了 Angular/Ionic 运行时 NgModule 应当做什么：

```
import { NgModule, ErrorHandler } from '@angular/core';
import { IonicApp, IonicModule, IonicErrorHandler } from 'ionic-angular';
import { MyApp } from './app.component';
import { HomePage } from '../pages/home/home';

import { StatusBar } from '@ionic-native/status-bar';
import { SplashScreen } from '@ionic-native/splash-screen';
```

```
@NgModule({
  declarations: [
    MyApp,
    HomePage
  ],
  imports: [
    IonicModule.forRoot(MyApp)
  ],
  bootstrap: [IonicApp],
  entryComponents: [
    MyApp,
    HomePage
  ],
  providers: [
    StatusBar,
    SplashScreen,
    {provide: ErrorHandler, useClass: IonicErrorHandler}
  ]
})
export class AppModule {}
```

有几个 Ionic 方面的问题值得注意。首先，是从 ionic-angular 库中导入 IonicApp 和 IonicModule。IonicModule 的 `forRoot` 方法用于指定 App 的根组件。我们可以传入一个 `config` 对象给 IonicModule。这个对象允许我们指定一些东西，比如返回按钮的文字、图标样式以及 tab 的位置。因为我们没有改变任何设置，我们将保留它们为对应平台的默认值。完整的配置选项请参考配置设置：*http://ionicframework.com/docs/api/config/Config/*。

我们还定义了 NgModule 的 `bootstrap` 参数，用 `IonicApp` 作为实际的引导进程。

对于 `entryComponents` 参数，我们指定了一个数组，包含了所有我们 App 中用到组件。例如这里的 `MyApp` 组件和 `HomePage` 组件。这样，AoT compiler（预先编译器）可以用于改善 App 性能。

然后，我们来看一下 *app.component.ts* 文件，在文件的开始是 `import` 语句。如果你记得的话，Angular 2 的依赖注入是这样处理的：

```
import { NgModule, ErrorHandler } from '@angular/core';
import { IonicApp, IonicModule, IonicErrorHandler } from 'ionic-angular';
import { MyApp } from './app.component';
import { HomePage } from '../pages/home/home';

import { StatusBar } from '@ionic-native/status-bar';
import { SplashScreen } from '@ionic-native/splash-screen';
```

第一个 `import` 语句用来从 Angular core 中加载 `Component` 模块。

### 关于 @App 组件

在 Ionic 2 早期版本，App 使用 App 模块进行引导。在 beta 8 中，Ionic 团队用普通的 Component 模块替换掉了 App 模块。

第二个 `import` 语句从 Ionic 框架库加载了 `Platform` 模块。第三个 `import` 语句从 Ionic Native 库加载 StatusBar 和 Splashscreen 插件。最后一个 `import` 语句加载我们的 `HomePage` 组件，这个组件是在 *pages* 目录下的 *home* 目录中定义的。我们会稍后介绍这个模块。

然后，才是 `@Component` 装饰器：

```
@Component({
  templateUrl: 'app.html'
})
```

### 什么是装饰器？

装饰器是一种简单的用于修改类、属性、方法和方法参数的方法。

在我们的 `Component` 装饰器中，我们指定了要渲染的模板。模板可以用两种形式来定义：行内，或者外部文件（通过 `templateUrl` 来引用）。

如果模板足够简单，我们可以使用行内模板。例如，*app.component.ts* 文件中就使用了以行内模板的形式定义的模板。它们通常会是这个样子：

```
template: `<ion-nav [root]="rootPage"></ion-nav>`
```

对于复杂的 HTML 模板，你可能就需要放到单独的文件中定义了。

我们来看一下 *app.html*。这个模板只有一个 `<ion-nav>` 组件：

```
<ion-nav [root]="rootPage"></ion-nav>
```

在 Ionic 1 中，这只是一个基本的导航容器，用于包含我们的内容。我们待会介绍 Navigation 组件。另外我们还设置了导航组件的根页面。我们使用了 Angular 2 的单向数据绑定语法 [] 将 root 属性设置为 `rootPage` 变量。回到 *app.component.ts* 文件，你可以在类定义中看到这个变量是怎么使用的：

```
export class MyApp {
  rootPage = HomePage;

  constructor(platform: Platform, statusBar: StatusBar, splashScreen:
  SplashScreen) {
    platform.ready().then(() => {
      // Okay, so the platform is ready and our plugins are available.
      // Here you can do any higher level native things you might need.
      statusBar.styleDefault();
      splashScreen.hide();
    });
  }
}
```

Ionic CLI 自动将这个类命名为 MyApp。然后，将 `rootPage` 变量设置为 HomePage 组件。因为使用的是 TypeScript，我们也需要指定变量的类型，在这里，我们设置它的类型为 any。

这个组件的构造器有一个 Ionic Platform 组件参数。在构造器中，`platform.ready` 函数返回一个 JavaScript Promise（承诺）对象。当 Ionic 和 Corodva 引导成功后，这个 Promise 会返回，任何与设备相关的代码都可以在此时调用了。

### DeviceReady 事件

如果你曾经做过 Cordova 或 PhoneGap 开发，你可能知道 deviceReady 事件，它会在 Web-to-native 桥完成初始化之后触发一次。它和传统 Web 开发中的 documentReady 事件类似。

在 Ionic 2 的早期版本中，我们调用一个 `ionicBootstrap`，但现在是通过 NgModule 函数来处理了。

*app* 目录的下一个文件是 *app.scss* 文件，这个文件是一个 App 全局的 Sass 文件。如果你不熟悉 Sass，可以把它看成是一种可以编译成 CSS 的样式表语言。后面会介绍样式和主题。现在，我们暂时略过这个文件。

最后一个文件是 *main.ts*。它是应用程序的入口点。我们的 App 默认是运行在开发模式。*main.ts* 文件看起来是这个样子：

```
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';

import { AppModule } from './app.module';
```

```
platformBrowserDynamic().bootstrapModule(AppModule);
```

首先加载 platformBrowserDynamic 模块，接着是 AppModule 模块。然后调用 bootstrapModule 函数，并传入我们的 AppModule。对于生产版的 *main.ts*，会加载 enableProMode 模块并在调用 bootstrap 之前调用它。

注意，这些文件负责 App 的初始化启动。你不需要关心这些文件，除非你需要覆盖配置或者修改全局样式。在我们需要测试或者发布 App 时，我们会添加一个参数来编译我们的 App，这会使用生产模式进行编译。

### pages 目录

Ionic CLI 还会生成一个 *pages* 目录。在 Ionic 1 中，通常是将 HTML 模板放到一个目录，而将对应的 *controller.js* 文件放到另一个目录。现在，有一个良好的改变，即将所有相关的文件放到一起。因此，在 pages 目录中，我们会找到一个 home 目录。在这个目录中，我们会发现 *home.html*、*home.css* 和 *home.ts*。当编写 Ionic 2 应用时，你会为每个页面创建一个目录，这些目录都位于 *pages* 目录下。

### home.html 文件

看一下这个文件的 HTML 代码，我们会发现所有的标签都以 <ion- 开始。原因之一就是 Ionic 框架是基于 Angular 框架构建的，它能够扩展 HTML 语言，加入一些表示移动组件的标签。这在 Ionic 2 中当然是理所当然的：

```
<ion-header>
  <ion-navbar>
    <ion-title>
      Ionic Blank
    </ion-title>
  </ion-navbar>
</ion-header>

<ion-content padding>
  The world is your oyster.
  <p>
    If you get lost, the <a href="http://ionicframework.com/docs/v2">
    docs</a> will be your guide.
  </p>
</ion-content>
```

为了让 <ion-navbar> 位于 content 内容之上，并在页面转换过程中保持固定位置，它必须放在 <ion-header> 标签之内。在 <ion-navbar> 标签中，我们用 <ion-

title> 标签设置 Navbar 的标题。<ion-title> 组件是平台相关的。在 iOS，标题文本会位于 Navbar 的正中，因为这是苹果 iOS 人机界面指南中要求的；但在 Android 或 WindowsPhone 中，标题文本是左对齐的。这是一个基本组件的例子，Ionic 在底层为实现平台相关的样式做了大量工作。

在 Navbar 下面是 <ion-content>。这个标签指定了主容器，它里面会用于放置 App 的绝大部分界面元素。它有一个 Angular 指令 padding，用于提供某种 CSS 对应样式给容器。对于空白模板创建的项目，<ion-content> 里面会有一串文本，一个 <p> 标签，一个 <a> 标签。这是一个只使用基本 HTML 来定义自己内容的极好的例子。

### home.ts 文件

这个 TypeScript 文件中代码很少。事实上，它只创建了一个最简单的空白的 Ionic 页面：

```
import { Component } from '@angular/core';
import { NavController } from 'ionic-angular';

@Component({
  selector: 'page-home',
  templateUrl: 'home.html'
})
export class HomePage {

  constructor(public navCtrl: NavController) {

  }
}
```

我们首先从 Angular 核心库导入了 Component 模块。和 App 组件被统一放到 Component 一样，Page 组件也被统一放到了 Component。然后我们从 Ionic Angular 库导入了 NavController 组件。在开发 Angular 2 时，有一件最重要的事情就是为了使用 Angular 和 Ionic 中的组件和功能，你必须导入它们。默认，不能再使用整个库。

Component 装饰器将 templateUrl 指定为 home.html，选择器指定为 page-home。

### build 目录

在 Ionic 2 的 RC 1 版本之前，Ionic CLI 会以稍微不同的方式来编译这个最终的运行版本的 App。它会将所有的 HTML 文件拷贝到 www 目录的 build 目录。任何对 Sass 文件（.scss 文件）的改变都会被编译并追加到每个平台的 css 文件中。TypeScript 文件会转译成 ES 5 并和 Angular 与 Ionic 代码一起打包，保存成 *app.bundle.js*。

虽然在 www 目录中仍然存在 build 目录，但里面却找不到 HTML 文件了。现在，Ionic CLI 会生成一个 App 的运行版本，HTML 模板会预编译并包含在 *main.js* 文件中。现在的 build 目录中会有 4 个文件：*main.css*、*main.js*、*main.js.map* 和 *polyfills.js*。所有 .scss 文件都被编译到 main.css。TypeScript 仍然会编译成 ES5，和 Angular、Ionic 代码一起打包到 *main.js*。*main.js.map* 文件在进行调试时会用到，*polyfill.js* 用于解决有可能出现的跨浏览器兼容性问题。

### theme 目录

*src* 目录中的第二个子目录是 Theme 目录。这里面会有一个 Sass 文件，我们可以通过修改这个文件来覆盖 Ionic 默认的 CSS 样式。这个文件我们就不花太多时间了，只是表明 Ionic 2 能够以非常简单的方式定制化 App 的可视化风格。

### Assets 目录

这个目录是从 RC 0 版本开始加入的。这个目录的目的是用于存放各种 App 中用到的资源文件，比如字体和图片。

### declarations.d.ts 文件

因为 TypeScript 使用了静态类型，我们需要能够“描述”我们想使用和导入的代码。这是以类型定义的方式进行的，TypeScript 团队管理了一个非常巨大的类型定义集合。如果因为某些原因导致你无法找到第三方库的类型，你可以在 *declarations.d.ts* 文件中创建一个简单的类型定义。*.d.ts* 扩展名表明这是一个定义文件而不是真正的代码。在这个文件中，我们可以添加一句代码来声明我们的模块：

```
declare module 'theLibraryName';
```

这句代码告诉 TypeScript 编译器该模块已存在，它是一个 any 类型的对象。将允许这个库正常使用而不会导致 TypeScript 编译器报错。

### manifest.json 文件

因为 Ionic 可以用于创建渐进式 Web App(PWA)，所以会包含一个 *manifest.json* 文件。我们会在第 13 章讨论这个文件和 PWA。

### service-worker.js 文件

*src* 目录中最后一个文件是 *service-work.js* 文件。它是 PWA 的另一个组成部分。我们现在可以忽略它，在第 13 章时再正式介绍它。

现在我们已经浏览了 Ionic CLI 模板创建的基本文件，接下来我们要在 Ionic2Do App 中修改这些文件。

## 修改页面结构

对于某些项目使用 Home 作为首页名称是可以的，但我想将我们的结构改成更能够反映页面功能的名称：这个页面是一个任务列表。我们需要看一下我们的文件和目录并将它们修改为和 tasklist 相关。我们将从 *app.component.ts* 文件开始：

```
import { Component } from '@angular/core';
import { Platform } from 'ionic-angular';
import { StatusBar } from '@ionic-native/status-bar';
import { SplashScreen } from '@ionic-native/splash-screen';

import { TaskListPage } from '../pages/tasklist/tasklist';

@Component({
  templateUrl: 'app.html'
})
export class MyApp {
  rootPage = TaskListPage;

  constructor(platform: Platform, statusBar: StatusBar, splashScreen:
  SplashScreen) {
    platform.ready().then(() => {
      // Okay, so the platform is ready and our plugins are available.
      // Here you can do any higher level native things you might need.
      statusBar.styleDefault();
      splashScreen.hide();
    });
  }
}
```

### Visual Studio Code

如果你的编辑器是 Visual Studio Code，你可能会看到在某些代码下面会有一些红色的波浪线。这是编辑器提示你代码可能有问题，比如文件缺少或者无法引用的变量。

我们还需要修改 *app.module.ts* 文件。因为修改了启动组件，我们需要修改这个文件以使它和新的组件名和目录保持一致：

```
import { NgModule, ErrorHandler } from '@angular/core';
import { IonicApp, IonicModule, IonicErrorHandler } from 'ionic-angular';
import { MyApp } from './app.component';
import { TaskListPage } from '../pages/tasklist/tasklist';

import { StatusBar } from '@ionic-native/status-bar';
import { SplashScreen } from '@ionic-native/splash-screen';

@NgModule({
  declarations: [
    MyApp,
    TaskListPage
  ],
  imports: [
    IonicModule.forRoot(MyApp)
  ],
  bootstrap: [IonicApp],
  entryComponents: [
    MyApp,
    TaskListPage
  ],
  providers: [
    StatusBar,
    SplashScreen,
    {provide: ErrorHandler, useClass: IonicErrorHandler}]
})
export class AppModule {}
```

保存这个文件，然后我们来调整目录结构和文件名。在 page 目录中，将 home 修改为 tasklist：[译注 1]

```
import { Component } from '@angular/core';
import { NavController } from 'ionic-angular';

@Component({
  selector: 'page-tasklist',
  templateUrl: 'tasklist.html'
```

译注 1：home.ts 文件中。

```
})
export class TaskListPage {

  constructor(public navCtrl: NavController) {

  }
}
```

切换到 *tasklist.scss* 文件，将选择器 page-home{} 修改为 page-tasklist{} 然后保存文件。

如果已经运行了 $ ionic serve 命令，在终端中用 Ctrl+C 先关闭它。然后运行 $ ionic serve，我们将看不到任何改变。

首先我们来修改 HTML 模板，打开 *tasklist.html*。

在 <ion-title> 标签中，将 Ionic Blank 修改为 Tasks。然后添加一个按钮到 header 中，这个按钮允许我们向任务列表中添加任务。将按钮添加到 navbar 或者 navbar 的父组件 toolbar 中，这和添加标准按钮是不一样的。

在这个标签中，我们需要添加一个属性，控制它所包含的按钮的位置。它有两个取值：start 或者 end。我想将 add item 按钮放在 header 的最右边，因此要用 end：

```
<ion-buttons end></ion-buttons>
```

然后，我们添加一个标准的 <button> 元素，并使用 Angular 2 语法指定它的单击事件处理函数为 addItem()。我们还使用一个 ion-button 指令，指定按钮样式和对应平台样式匹配。在 Ionic 2 中还有一个改变就是 icon-left 和 icon-right 指令。这两个指令会在图标的左边或右边添加一个小间距。否则图标和文字会紧挨在一起。如果你的按钮上只有图标，可以用 icon-only。这会在图标两边都添加适当的空白间距：

```
<ion-header>
  <ion-navbar>
    <ion-title>
      Tasks
    </ion-title>
    <ion-buttons end>
      <button ion-button icon-left (click)="addItem()">Add Item</button>
    </ion-buttons>
  </ion-navbar>
</ion-header>
```

保存文件。如果 `$ ionic serve` 正在运行，我们会看到浏览器中已经显示出我们的修改（见图 7-3）：

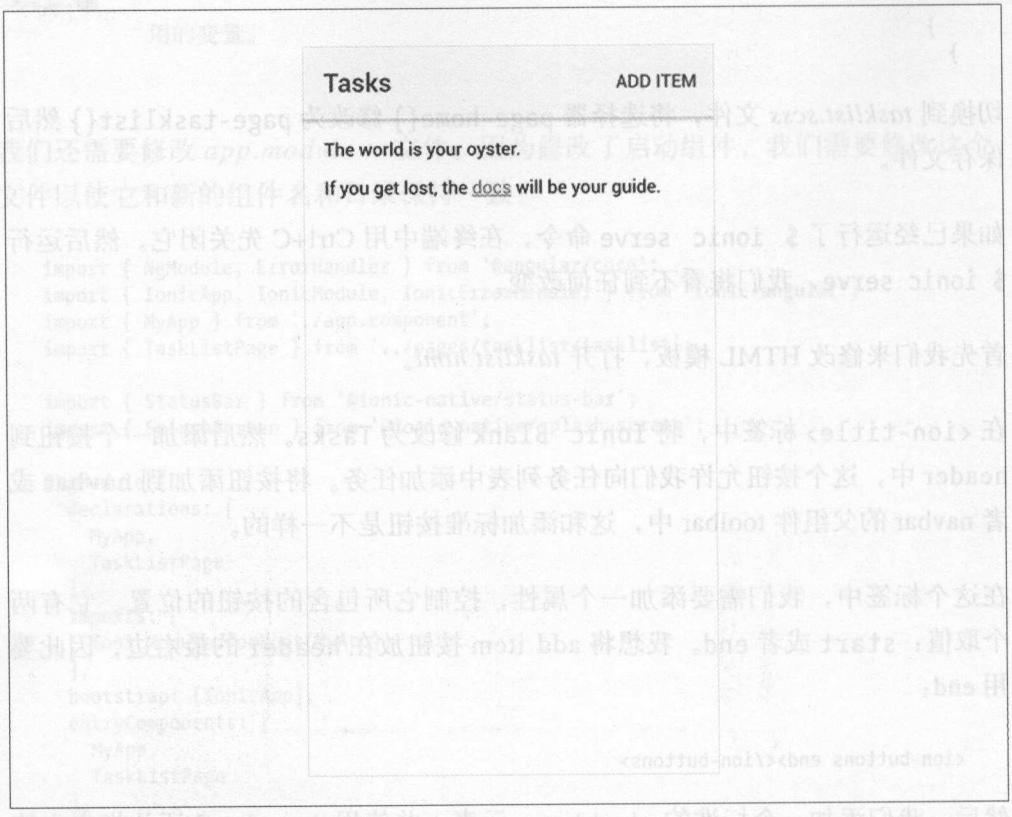



图 7-3：修改 Ionic2Do 的 header

Ionic 封装了大量图标。它们实际上被放到了另外一个 GitHub 项目里，因此只要你想，你可以在 Ionic 之外的项目中使用它们。

Ionic 2 在 Ionicon 库的使用上做了几个改变。首先，我们现在要使用 `<ion-icon>` 标签，而不是 `<i>` 标签。然后，不再通过 CSS 类的方式来引用我们想用的图标，而是通过 `name` 属性来指定我们想用的图标。对于 Add Item 按钮，一个标准的加号图标就可以了。这个标签写成这个样子：

```
<button ion-button icon-left (click)="addItem()">
  <ion-icon name="add"></ion-icon> Add Item
</button>
```

许多图标都有两种版本：材料设计和iOS。Ionic会自动根据不同平台来使用正确的版本。

但是，如果你需要自己来控制，则需要显式地为每个平台指定图标。可以通过md（材料设计）属性和ios属性来指定某个平台的图标：

```
<ion-icon ios="logo-apple" md="logo-android"></ion-icon>
```

如果你想使用特定的图标，只需要使用图标的完整名。例如，如果你想在所有平台上都使用iOS的地图图标，你可以这样写：

```
<ion-icon name="ios-map-outline"></ion-icon>
```

要查找完整图标名，请打开Ionic Framework Ionicons主页（*http://ionicons.com/*），然后选择一个图标，该图标的信息会显示出来（见图7-4）：

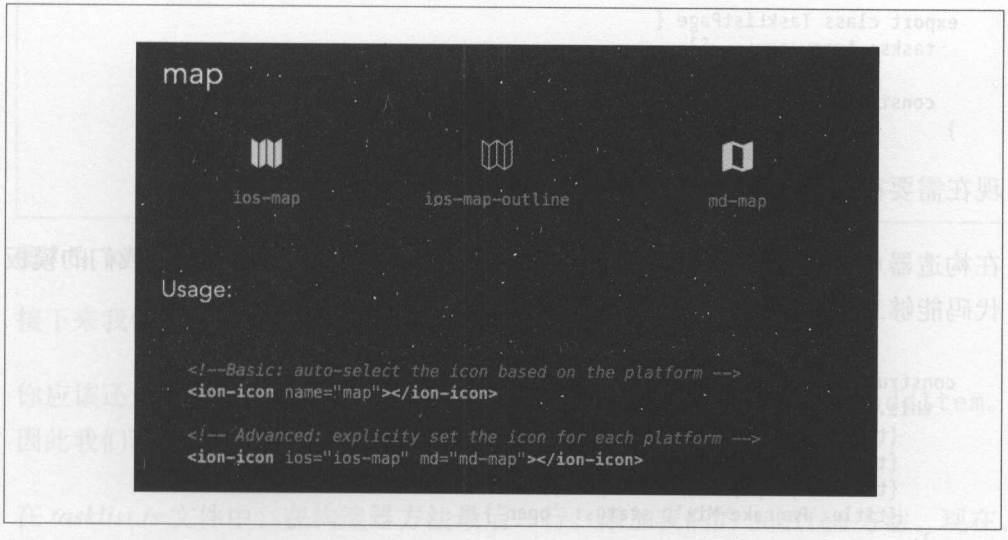



图7-4：地图图标的完整图标信息

现在，让我们回到任务列表的创建上。

将<ion-content>中的内容替换成：

```
<ion-content>
  <ion-list>
    <ion-item *ngFor="let task of tasks">{{task.title}}</ion-item>
  </ion-list>
    </ion-content>
```

我们来看一下这些代码。<ion-content>标签充当内容容器，如果内容大小超过了 viewport 大小，它会自动允许滚动。我们使用了 <ion-list>，你可能猜到了，它用于展示多行信息，这里它会显示多行待办事项。在 <ion-list> 中，我们定义了列表项的模板，以及如何映射和绑定数据。

在 Angular 1，我们应该用 ng-repeat 指令来指定一个数组以便循环生成我们的待办事项。Angular 2 将这个指令变成了 *ngFor。它们的功能是一样的。在我们的例子中，我们会遍历 tasks 数组，并将它的每个元素赋给一个本地变量 task。通过数据绑定语法 {{task.title}}，我们会渲染每个任务的标题字符串。保存文件，打开 tasklist.ts 文件。

在这个类中，我们需要定义 tasks 变量。因为我们使用的是 TypeScript，需要指明类型。这里，可以将 tasks 数组定义为一个 any 对象的数组：

```
export class TaskListPage {
  tasks: Array<any> = [];

  constructor(public navCtrl: NavController) {
}
```

现在需要在类定义中实现一个真正的构造器（constructor）函数。

在构造器中，将我们的 tasks 数组初始化为一些临时数据，以便让我们的模板代码能够工作：

```
constructor(public navCtrl: NavController)  {
  this.tasks = [
     {title:'Milk', status: 'open'},
     {title:'Eggs', status: 'open'},
     {title:'Syrup', status: 'open'},
     {title:'Pancake Mix', status: 'open'}
  ];
}
```

保存这个文件。再一次使用 $ ionic serve 命令，在浏览器中预览 App（见图 7-5）：

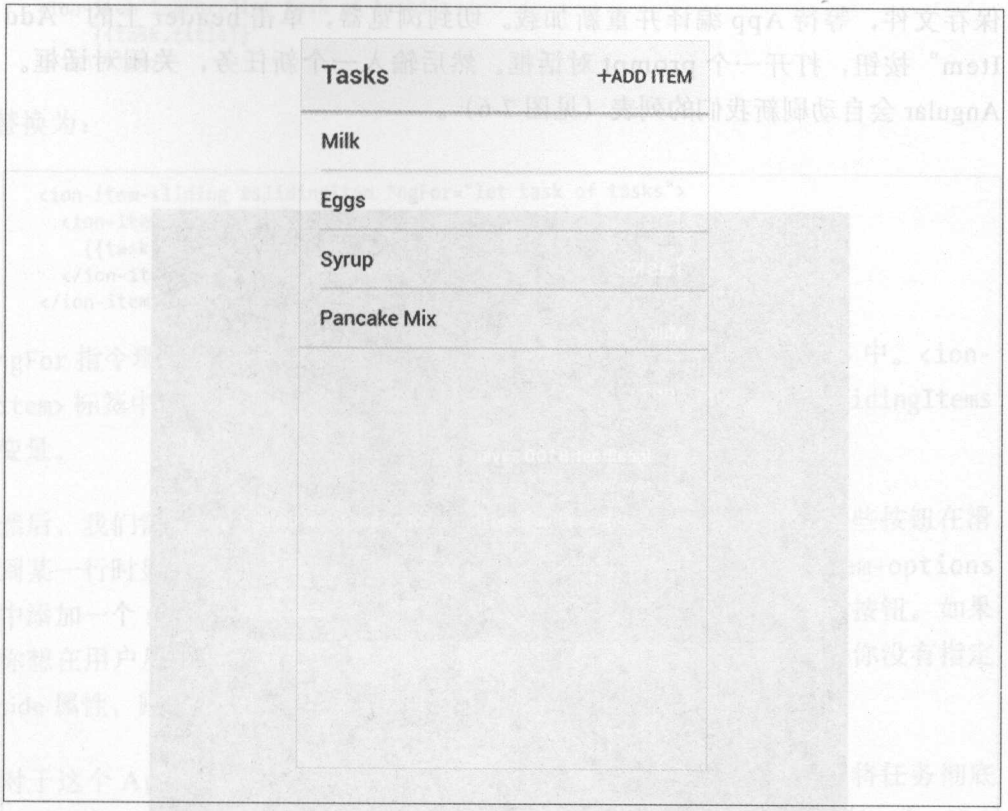



图 7-5：Ionic2Do App

接下来我们要添加一个待办到任务列表中。

你应该还记得，我们曾经在 HTML 中指定了一个单击事件处理方法 `addItem`。因此我们现在要实现这个方法。

在 *tasklist.ts* 文件中，在构造器方法最后一行，添加我们的 `addItem` 方法。现在，我们将用标准的 `promt` 方法显示一个对话框，允许用户输入新任务的标题。然后将用户输入的字符串封装到一个普通对象中，再将对象 push 进 `tasks` 数组：

```
addItem() {
  let theNewTask: string = prompt("New Task");
  if (theNewTask !== '') {
    this.tasks.push({ title: theNewTask, status: 'open' });
  }
}
```

保存文件，等待 App 编译并重新加载。切到浏览器，单击 header 上的“Add Item”按钮，打开一个 prompt 对话框。然后输入一个新任务，关闭对话框。Angular 会自动刷新我们的列表（见图 7-6）。

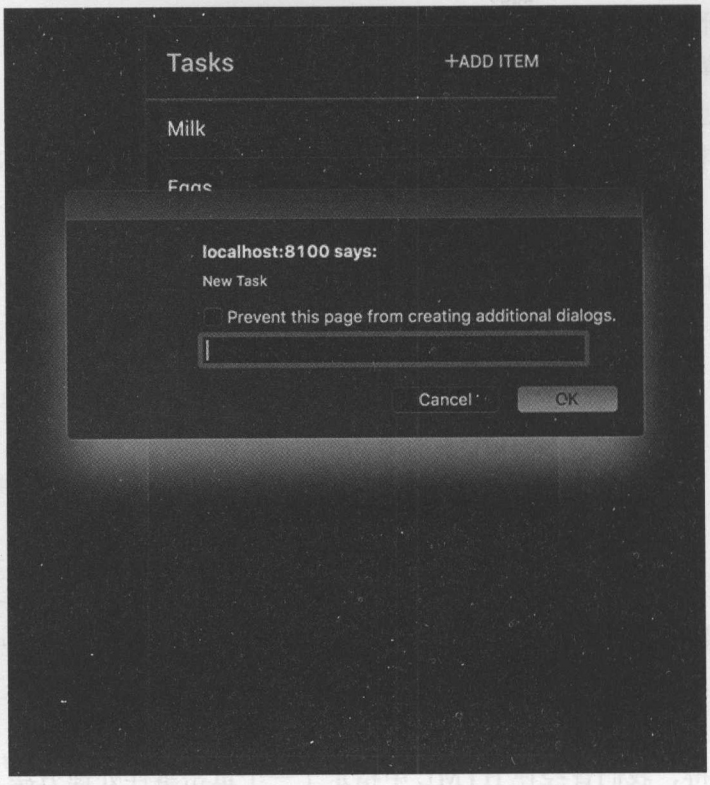



图 7-6：Ionic2Do 添加新任务对话框

如果你打开预览时用的是 `$ ionic serve --lab` 命令，你会注意到只有一个列表刷出了新加的任务。这是因为 `ionic serve` 实际上运行了 2 个 App 实例。

我们现在已经可以向任务列表中添加东西了，让我们来添加一个方法来表示任务已完成。一个常见的方式是在某一行上通过向左滑动来显示一系列按钮。Ionic 2 提供了一个 `<ion-item-sliding>` 组件来实现这种体验。

将这段代码：

```
<ion-item *ngFor="let task of tasks">
    {{task.title}}
</ion-item>
```

替换为：

```
<ion-item-sliding #slidingItem *ngFor="let task of tasks">
  <ion-item>
    {{task.title}}
  </ion-item>
</ion-item-sliding>
```

ngFor 指令现在放到了 <ion-item-sliding> 组件而不是 <ion-item> 中。<ion-item> 标签中的元素将每一行的内容显示为可见。我们还需要引用 slidingItems 变量。

然后，我们需要用 <ion-item-options> 组件来包含我们的按钮，这些按钮在滑到某一行时显示。这个组件支持左滑、右滑和双向滑动。在 ion-item-options 中添加一个 side='right' 表示我们需要在用户从右向左滑动时显示按钮。如果你想在用户从左向右滑动时显示按钮，则使用 side='left'。如果你没有指定 side 属性，则默认为 right。

对于这个 App，我们会用一个按钮标记任务已完成，用另一个按钮将任务彻底从列表中删除。这只需要使用标准的 <button> 标签。每个按钮都有一个 click 函数，以及一个 Ionicon 库中的图标。代码如下：

```
<ion-list>
  <ion-item-sliding *ngFor="let task of tasks">
    <ion-item>
      {{task.title}}
    </ion-item>
    <ion-item-options side="right">
      <button ion-button icon-only (click)="markAsDone(task)" color="secondary">
        <ion-icon name="checkmark"></ion-icon>
      </button>
      <button ion-button icon-only (click)="removeTask(task)" color="danger">
        <ion-icon name="trash"></ion-icon>
      </button>
    </ion-item-options>
  </ion-item-sliding>
</ion-list>
```

要将一个任务标记为已完成，我们的 click 事件处理器会调用一个 markAsDone 函数，并传递一个该行所对应的任务进去。如果之前使用过 Angular，你会知道

这是这个框架提供的一种能力。你可以让 Angular 为你记住每一行并让它解释我们正在交互的任务是什么，而不是让我们自己来记住这些。

对于按钮的显示内容，我们只需要使用 `<ion-icon>` 组件。现在，因为在 `<ion-icon>` 标签中并没有实际内容，你可能会想使用自关闭语法。但 Angular 要求这些标签必须用规范写法而不能使用自关闭语法（见图 7-7）。

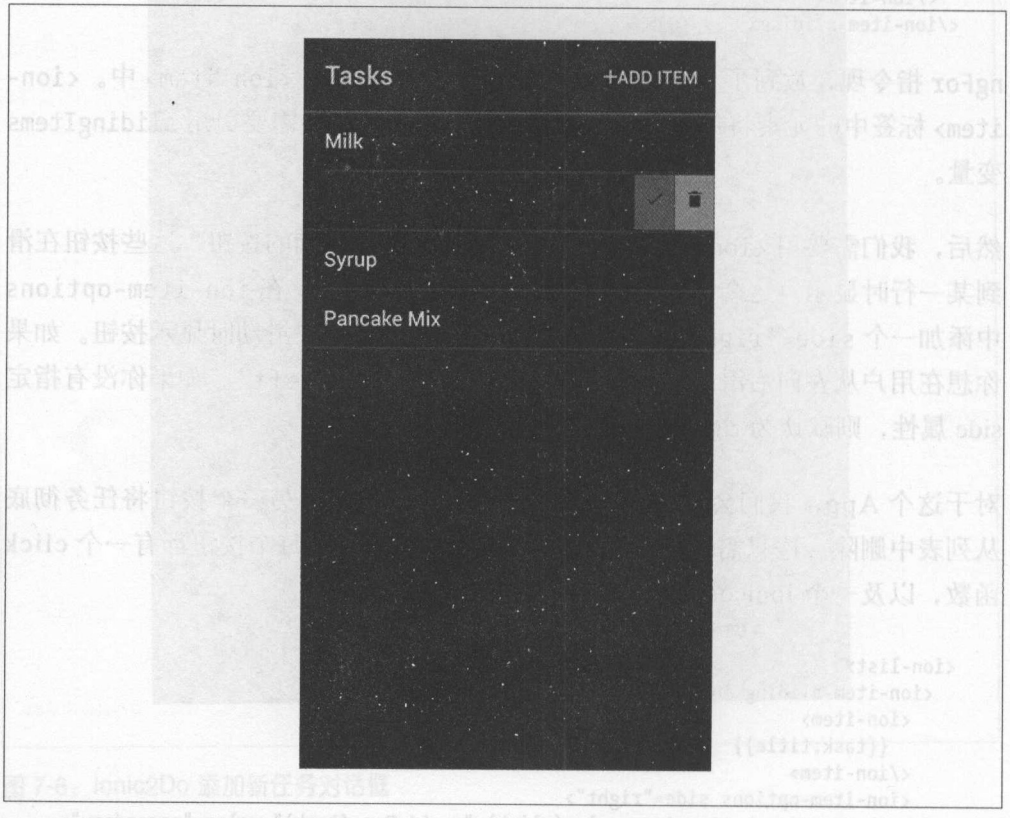



图 7-7：Ionic2Do 的滑动列表按钮

让我们回到 *tasklist.ts* 文件，在 `addItem` 函数后面添加两个函数：`markAsDone` 和 `removeTask`：

```
markAsDone(task: any) {
    task.status = "done";
}
removeTask(task: any) {
    task.status = "removed";
```

```
    let index = this.tasks.indexOf(task);
    if (index > -1) {
      this.tasks.splice(index, 1);
    }
  }
```

先别忙着测试你的 App。我们需要在用户将任务标记为已完成之后，在这个任务上画一条删除线。使用 CSS 我们可以很容易做到这一点，我们可以使用 `text-decoration` 属性。在 *tasklist.scss* 文件，添加下列 CSS：

```
.taskDone {text-decoration: line-through;}
```

Angular 提供了一个能够根据不同条件来将 CSS 类应用到某个元素上的方法。我们准备让 Angular 在任务的状态属性 status 为 done 时，将 `taskdone` 这个 CSS 类应用到 `<ion-item>` 上：

```
<ion-item [ngClass]="{taskDone: task.status == 'done'}" >
```

### Angular 1 到 Angular 2

这个指令很好地演示了 Angular 1 和 Angular 2 之间的一些微妙改变。在 Angular 1 中，对应的语法是 `ng-class= "expression"`。在 Angular 2 中，变成了 `[ngClass]= "expression"`。

保存所有文件，在浏览器中预览你的 App。现在你可以滑动某一行，显示出两个按钮。单击勾选按钮，将任务标记为已完成。你可以发现字符串上多出来的删除线。但是，当单击完选项按钮后，列表行无法滑回原样。我们来搞定这个界面问题。

在 *tasklist.ts* 文件中，首先修改 `import` 语句，将 `ItemSliding` 组件也包含进去：

```
import {NavController, ItemSliding} from 'ionic-angular';
```

然后我们要修改 `<ion-item-sliding>` 组件。添加一个本地的临时变量 `#slidingItem`。`<ion-item-sliding>` 组件用这个变量来引用每一行。然后，将这个变量作为第一个参数传递给两个按钮的 click 事件处理方法。修改后的代码如下：

```
<ion-item-sliding *ngFor="let task of tasks" #slidingItem>
  <ion-item [ngClass]="{taskDone: task.status == 'done'}">
```

```
      {{task.title}}
    </ion-item>
    <ion-item-options side="right">
      <button ion-button icon-only (click)="markAsDone(slidingItem, task)"↵
      color="secondary">
        <ion-icon name="checkmark"></ion-icon>
      </button>
      <button ion-button icon-only (click)="removeTask(slidingItem,task)"↵
      color="danger">
        <ion-icon name="trash"></ion-icon>
      </button>
    </ion-item-options>
  </ion-item-sliding>
```

保存文件，打开 *tasklist.ts* 文件。我们需要为 `markAsDone` 函数和 `removeTask` 函数中增加新的参数。这个参数是一个 ItemSliding 类型：

```
markAsDone(slidingItem: ItemSliding, task: any) {
  task.status = "done";
  slidingItem.close();
}

removeTask(slidingItem: ItemSliding, task: any) {
  task.status = "removed";
  let index = this.tasks.indexOf(task);
  if (index > -1) {
    this.tasks.splice(index, 1);
  }
  slidingItem.close();
}
```

我们可以调用 slidingItem 的 `close` 方法，将我们的 list item 滑回原样。

### Ionic 1 到 Ionic 2

Ionic 1 也有一个类似的组件。但是，触发 close 动作必须注入和使用一个 `$ionicListDelegate` 对象。Ionic 2 对代码进行了简化，现在只需要引用列表中的行就可以了。

测试一下新的 App，你会发现当我们单击两个按钮之后，列表行会滑回原处。

## 添加全扫手势

你可能在操作过列表行时，使用过全扫手势。一个很好的例子就是 iOS 的 Mail 程序。如果你在列表上做出一个持续较短的轻扫动作，选项按钮会显示。这和

我们在自己的 App 所做的是一样的。但在 iOS 的 Mail App 中，如果你继续轻扫动作，最后一个选项按钮将被自动单击。让我们在 App 中实现这个功能。

首先，我们需要添加一个事件监听器，监听 `ionSwipe` 事件。这个事件会在用户操作一个全扫（full-swipe）手势时触发。我们还需要指定这个事件的处理函数。通常，它应该和最后一个选项按钮的事件处理函数是同一个。因此，我们的 `<ion-item>` 变成了：

```
<ion-item-options side="right" (ionSwipe)="removeTask(slidingItem, task)">
```

接着，我们需要在按钮上添加另外一个属性，用于在这个手势被操作时，能够看到拉伸效果。因为这动作是在最后一个按钮上进行的，我们将这个属性添加到删除任务的那个按钮上。新的 <button> 是这个样子的：

```
<button ion-button icon-only expandable color="danger" ↵
  (click)="removeTask(slidingItem, task)">
```

通过这两个额外的工作，我们就在我们的 App 中添加了对全扫手势的支持。

## 简单主题

除了让这些组件的外观和平台匹配，我们还简单地修改了它们的颜色。Ionic 有 5 种内置的主题颜色：primary（蓝色）、secondary（绿色）、danger（红色）、light（浅灰色）和 dark（深灰色）。这些主题颜色，可以直接添加到大部分 Ionic 组件中使用。让我们来为 header 添加一点色彩，设置它的主题色为 `primary`：

```
<ion-navbar color="primary">
```

我们曾经为每个选项按钮设置过主题色：

```
<button color="secondary" (click)="markAsDone(slidingItem, task)">
    ...
<button color="danger" (click)="removeTask(slidingItem, task)">
```

我们在后面的章节中讨论更多关于设置 Ionic App 样式的内容。现在，我们的 App 已经有一些颜色了。

# 正确地声明类型

Ionic 的 build 脚本在类型检查方面是非常严格的。因此 tasks 变量被声明为 any 而不是 Object。现在可以在一个 Oject 中提供一些嵌套的类型声明。以我们的任务为例，它可以定义为：

```
tasks: Array<{ title: string, status: string }> = [];
```

但如果我们在 maskAsDone 和 removeTask 函数中将 task 参数的类型修改为 Object，编译时会报错：

```
Typescript Error
Property 'status' does not exist on type 'Object'.
```

TypeScript 告诉我们，我们添加了一个 status 属性在 Task 对象上。在早期的 Ionic 编译脚本上，我们的 App 仍然可以工作；如果用现在的 build 脚本，我们需要解决这个问题。

解决办法非常简单：我们需要创建一个自定义的 Task 类，用它来替换泛型的 Oject。我们可以直接在 *tasklist.ts* 文件中定义 Task 类；但我们决定在 *task-list.ts* 文件同一目录下创建一个新文件 *task.ts*。在这个文件中，我们 export 一个类 Task。它有两个属性：title 和 status，两个都是 sting 类型：

```
export class Task {
  title: string;
  status: string;
}
```

在 *tasklist.ts* 文件中，我们需要导入这个类：

```
import { Component } from '@angular/core';
import { NavController, ItemSliding } from 'ionic-angular';
import { Task } from './task';
```

然后将 task 数组的类型从：

```
tasks: Array<Object> = [];
```

修改为：

```
tasks: Array<Task> = [];
```

我们还需要修改 `markAsDone` 函数和 `removeTask` 函数的参数。将 task 变量的类型从 any 修改为 Task。

## 保存数据

你可能注意到我们的 App 有一个很大的缺陷。我们添加的任务无法被保存。如果你重新打开 App，只会显示最初的 4 个任务。这个问题有几个解决办法。我们可以用 localStorage 保存数据，但这样我们的数据有可能被用户或者系统清除掉。WebSQL 也是一种方案，但这个规范已经不再维护了，因此它的长期支持会是一个问题。如果你想用 Cordova 插件，那么还有几个别的选择。因为我们的数据结构非常简单，可以简单文本的形式将它们保存到文件系统，然后进行读取。这不是很好的解决方案（也不安全），但也是一种办法。更好的选择是使用 SQLite 插件，还可以加上 PouchDB。我们还可以用 Ionic 自身提供的新的 Storage 模型。

但这些只是一对一的解决方案。当我们面对互联的、多设备的、多平台的世界时，我需要将我的任务列表在我所有的移动设备之间共享和访问。

## 创建 FireBase 账号

你可以有一个快捷的基于云存储的选择，这就是 Google 的 FireBase。FireBase 服务提供了一个实时的 JSON 数据库，可以用于存储和同步你的 App 数据。如果你还没有 FireBase 账号，你可以在 Firebase（*https://firebase.google.com/*）注册一个免费的开发者账号。

然后根据提示创建一个新项目。将项目名称设为 Ionic2Do。然后选择你的国家或地区，单击“Create Project”。这会创建一个基本的 Firebase 项目。在 dashboard 中，单击“Database”，这将显示我们默认的数据库。

要用我们的 App 连接上 Firebase，需要配置几个东西。Firebase 使这个工作变得非常简单：找到 Add Firebase to your Web app 按钮，单击它。这会显示一个连接我们 App 的完整代码。当然，我们只对这几个值感兴趣：

```
var config = {
  apiKey: "your-api-key",
  authDomain: "your-authdomain",
```

```
    databaseURL: "https://someurl.firebaseio.com",
    storageBucket: "someurl.appspot.com",
    messagingSenderId: "your-sender-id"
};
```

将这些代码保存到临时文件里，以备后面使用。

在使用这个数据库之前，我们需要修改一下安全设置。单击 Rules 标签，打开 Rules 编辑器。默认，数据库读写权限都需要进行权限认证。因为我们这个 App 仅仅是一个教程，我们可以放宽这个设置。将 rules 从：

```
{
  "rules": {
    ".read": "auth != null",
    ".write": "auth != null"
  }
}
```

修改为：

```
{
"rules": {
    ".read": "auth == null",
    ".write": "auth == null"
  }
}
```

然后单击“Publish”。现在，我们的数据库现在不需要权限验证就可以进行读 / 写了。

## 安装 Firebase 和 AngularFire 2

为了使用 Firebase，我们需要安装几个 Node 模块到我们的项目中。

### Beta 版说明

无论 AngularFire 还是 Firebase 库，目前都没有开发完成。这里列出的示例代码有可能随这些库而改变。请随时关注 O'Reilly 官网（*http://shop.oreilly.com/product/0636920044710.do*）或本书官网（*http://ionic2book.com/*）上的更新。

我们需要安装的第一个 Node 模块是 @types 模块。在 TypeScript 开发初期，对 *.d.ts*

文件有不同的类型定义管理器。到了 TypeScript 2.0，推荐的方法是用 @type 模块管理所有的其他库的定义：

```
$ npm install @types/request --save-dev --save-exact
```

然后在我们的项目中安装 Firebase。打开命令行：

```
$ npm install firebase --save
```

尽管我们可以直接使用 Firebase，AngularFire 库却能使这个过程变得简单。如果使用 Angular 2 的 Observables，和 Firebase 打交道就变得更简单了。如果你用过 Angular 1 和 Firebase，你可能会用 AngularFire 库将 AngularJS 绑定到 Firebase。AngularFire 2 是这个库的 TypeScript 版本。这个版本在编写本书时还是 beta 版。要安装 AngularFire 库，请使用下列命令：

```
$ npm install angularfire2 --save
```

## Ionic 编译系统

当我们用 Ionic CLI 编译 App 时，它会执行一系列脚本集，比如 Ionic App Scripts。Ionic App Scripts 最近已经从主框架中分离出来，变成了一个独立的 GitHub 库，因此这个库可以被单独更新和改进。这些脚本会将我们 App 中用到的各种文件、第三方库、资源以及其他东西打包成可执行的应用程序。在 Ionic 2 的开发过程中，开发团队 4 次修改了编译系统，从 Browserify 到 WebPack，再到 Rollup.js，然后又回到 Web Pack 2。我们建议读一下 Ionic 的官方博客（*http://blog.ionic.io/*），以了解最新的 Ionic 编译系统。

要安装最新的 Ionic App Scripts，在 Ionic 项目目录下运行命令：

```
$ npm install @ionic/app-scripts@latest
```

默认，Ionic App Scripts 会进行以下工作：

- 将源文件转译成 ES5 JavaScript。
- 对模板进行预先（AoT）编译。
- 对模板进行即时（JiT）编译。

- 在 JiT 编译时将模板植入。
- 为了加快运行时执行，对模块打包。
- 删除无用的组件和代码。
- 从打包的 Sass 文件生成 CSS。
- 自动将 vendor 添加到 CSS 前缀。
- 压缩 JavaScript 文件。
- 压缩 CSS 文件。
- 拷贝 src 静态资源到 *www*。
- 用 Lint 优化源文件。
- 监听源文件以便实时刷新。

和 Grunt 或 Gulp 这些外部任务执行器不同，这些脚本是通过 npm 脚本来执行的。这些脚本会为我们完成各种任务，见表 7-2。

表 7-2：任务列表

| 任务 | 描述 |
| --- | --- |
| build | 创建一个完整的 App build，默认使用 `development` 设置，可以用 `--prod` 创建优化的 build |
| clean | 清空 *www/build* 目录 |
| cleancss | 用 CleanCss 压缩生成的 CSS |
| copy | 执行拷贝，默认将 src/assets/ 和 src/index.html 文件拷贝到 www 目录 |
| lint | 根据 *.ts* 源文件和根目录下的 *tslint.json* 配置文件来运行 linter 命令 |
| minify | 对生成的 JS 包和编译好的 CSS 进行压缩 |
| sass | 创建一个所用到的模块的 Sass 汇编，在 Sass 进行汇编之前，必须至少运行过一次编译 |
| watch | 对开发模式下的 build 进行监听 |

除了这些任务外，App 脚本还支持丰富的定制，以便满足你的需要。可以定制的内容包括对各种配置文件、设置值、Ionic 环境变量都能够进行设置。具体可参考 GitHub 上（*http://bit.ly/2n1A4e*）关于 Ionic App Scripts 的相关内容。

## 将 AngularFire 添加到 app.module.ts 文件

为了使用 AngularFire/Firebase，需要修改我们的 *app.module.ts* 文件。首先，我们需要从 angularfire2 包中导入具体的组件：

```
import { AngularFireModule } from 'angularfire2';
```

第二个需要修改的地方是指定默认的 Firebase 配置。在 *@NgModule* 声明之前，添加如下代码（将其中的值替换成你的 Firebase 账号的值）：

```
export const firebaseConfig = {
  apiKey: "your-api-key",
  authDomain: "your-authdomain",
  databaseURL: "https://someurl.firebaseio.com",
  storageBucket: "someurl.appspot.com",
  messagingSenderId: "your-messageSenderID"
};
```

最后一个需要修改的地方是在 `@NgModule` 定义中的 `imports` 数组。在这里我们需要调用 `AngularFireModule` 并用我们的配置进行初始化。

```
imports: [
  IonicModule.forRoot(MyApp),
  AngularFireModule.initializeApp(firebaseConfig)
],
```

这样，我们就可以开始使用 AngularFire/Firebase 了。接下来我们将注意力放到 *tasklist.ts* 文件中来。

## 使用 Firebase 数据

和我们在 *app.module.ts* 中的修改类似，第一件事情就是修改 `import` 语句。我们需要从 AngularFire 2 库中导入 `AngularFire` 和 `FirebaseListObservable`：

```
import { AngularFire, FirebaseListObservable } from 'angularfire2';
```

一般，在和动态数据打交道时，一种常见的方法是使用 RxJS 库的 Observable 模块。如果你之前没接触过 RxJS 库，那么它是一套结合了异步编程和事件编程的代码库，它使用了 observable 集合和 JavaScript 中的 Array#extras 风格的写法。这个库其实是由 Microsoft 在维护着的。具体你可以在 GitHub 看到这个项目（*https://github.com/Reactive-Extensions/RxJS*）。

AngularFire 使用了 `Observable` 并将它扩展为一个定制化的版本 `FirebaseListObservable`。我们也会在我们的 App 中用到它。

然后，我们将用于保存任务的本地数组替换成来自 Firebase 的数据。

因此将 `tasks:Array<Task> = [];` 改成 `tasks:FirebaseListObservable<any[]>;`。这会让 tasks 变量支持修改。

我们的组件将使用 AngularFire 库和 `Firebase` 数据库打交道。为此，我们必须在构造器函数中增加一个参数，用于将这个库的引用传递到方法中。

```
constructor(public navCtrl: NavController, public af: AngularFire) { ... }
```

因为我们的任务列表将存储到远程，我们可以将 `tasks` 数组的内容替换成 AngularFire 请求 *tasks* 子目录后返回的数据。

```
this.tasks = af.database.list('/tasks');
```

这时，这个数组应该是空的，因为数据库里面还是空的。在使用 `$ ionic serve` 命令测试我们的 App 之前，我们需要在 *tasklist.html* 中添加一个管道操作到我们的 `ngFor` 指令上。为了能够使用 `FirebaseListObservable` 数组，我们需要告诉 Angular 数据是异步抓取的。因此，我们需要加入 `async` 管道。否则，当 Ionic/Angular 准备渲染模板时，就会出现没有数据并导致错误的情况。而添加 async 管道后，Angular 就会正确处理这些延迟数据：

```
<ion-item-sliding *ngFor="let task of tasks | async" #slidingItem
  (ionSwipe)="removeTask(slidingItem, task)">
```

### 什么是管道？

管道是一种函数，用于转换模板中的数据。Angular 提供了几种内置的管道用于转换字符大小写或者格式化数字。

你还可以向任务列表中添加新的条目。当你添加完新的条目之后，关闭服务器，再次打开。你会发现你添加的内容会在列表中显示。

此外，你还可以登进 Firebase 账号查看你的数据集。

你或许发现了，我们没有修改 addItem 函数。因为我们的 tasks 变量是一个绑定到我们的 Firebase 数据库的 FirebaseListObservable 对象，添加新的条目是通过 push 方法来进行的。

但是，为了能够和 Firebase 正确交互，我们还是要修改 markAsDone 方法和 removeTask 方法。我们不能直接和某个数组元素打交道了，而必须用 AngularFire 的方式来替代。

在 markAsDone 函数中，将：

```
task.status = "done";
```

替换为：

```
this.tasks.update(task.$key, { status: 'done' });
```

当 task 被添加到数据库时，task.$key 用于表示 Firebase 生成的唯一 key。

在 removeTask 函数中，将：

```
task.status = "removed";
let index = this.tasks.indexOf(task);
if (index > -1) {
  this.tasks.splice(index, 1);
}
```

替换为：

```
this.tasks.remove(task.$key);
```

保存文件，用 $ ionic serve 运行我们的 App。我们的滑动按钮能够正确地更新 Firebase 数据库。

你可能会看到在 task.$key 旁边会出现一个警告：'$key' does not exist on type 'Task' any。这个问题很容易搞定。打开 *task.ts* 文件，添加一个新属性 $key，设置它的类型为 any。

这样，只用了聊聊几行代码，我们就将本地版本的 to-do App 改成了“云版”的 App。我们稍微领略了一下 Firebase 和 AngularFire 2 的强大之处。建议你花一点时间熟悉一下这两个库的功能。

## 使用 Ionic Native

让我们在模拟器中预览一下 App。在命令行中，使用 `$ ionic emulate ios` 或 `$ ionic emulate android` 命令。

### 模拟 Android

我们曾经说过，默认的 Android 模拟器启动非常慢。我推荐用 Genymotion 作为替代。

### 模拟 iOS

当你运行 `ionic emulate iOS`，你可以指定具体的设备类型和 OS 版本。这需要在命令后添加 `--target="devicename, OSType"`。例如，如果想在 iPhone 5S、iOS 10 上测试，命令应该使用 `ionic emulate ios --target="iPhone-5s, 10.0"`。

我们的 App 应该能够连接到 Firebase 数据库并显示我们的任务列表。现在，让我们添加一个新任务并单击“Add Item”按钮（见图 7-8）。

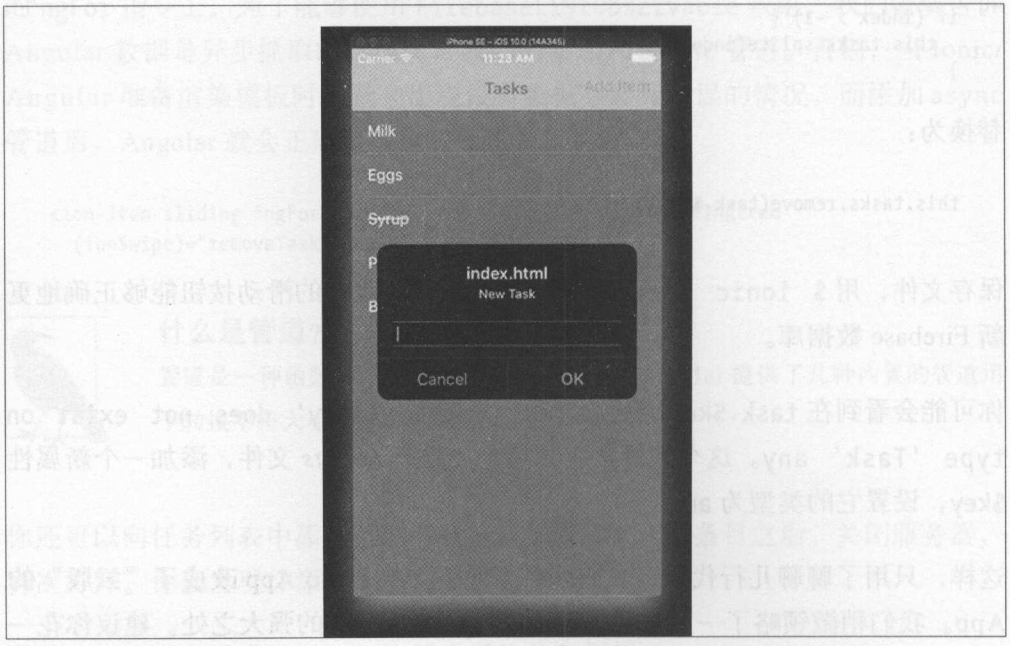



图 7-8：Ionic App 中显示的 JavaScript 对话框

注意这个对话框：这个 JavaScript 对话框和我们在浏览器中看到的 JavaScript 对话框是一样的。如果想将它显示得和本地 App 一样，我们必须在发布到 App 商店之前解决这个问题。

Cordova 有一个现成的插件提供了用于取代 JavaScript 对话框的本地对话框。为了在 Ionic 2 框架中使用 Cordova 插件，他们为绝大部分常用插件提供了封装。

默认，Ionic Native 是已经包含的，但如果你需要手动导入 Ionic Native 到项目中，可以用以下命令：

```
$ npm install @ionic-native/core --save
```

安装好 Ionic Native 之后，我们就可以拥有和 Cordova 插件进行交互的接口了。

### 关于 ngCordova

ngCordova 库（也是由 Ionic 团队进行维护）是用于 Angular 1 和 Ionic 1 项目的。Ionic Native 则是用于 Angular 2 和 Ionic 2 项目的。

接下来我们实际安装需要用到的插件。其中一个就是 Dialogs 插件。这个插件会显示一个原生对话框，而不是 Web 的对话框。在终端中，安装 Dialogs 插件的 Ionic CLI 命令是：

```
$ ionic plugin add cordova-plugin-dialogs
```

这会下载所有已安装的平台的插件代码并修改 *config.xml* 文件，以便使用相关的代码来编译 App。

### 插件源码

在过去，插件以 org.apache.cordova.* 命名空间来进行引用。当插件迁移到 npm 之后，命名空间就改成了 cordova-plug-*。在某些文档中你还会看到旧的命名系统，因此你需要将它转换成 npm 的命名系统。

从 Ionic Native 3.x 开始，要在我们的 App 使用这个插件需要多加一个步骤。在 Ionic Native 2.x 时候，整个 Ionic Native 库会被加载到你的 App（只是接口，不是真正的插件）中。现在，每个插件都能够选择性地添加到 App 中，这将让你

的 App 变得更小，让启动速度变得稍微快上那么一点。这种改变体现插件接口的使用方式上，也就是我们必须在 `@NgModule` 的提供者列表中 import 和添加每个插件提供者。

在 *app.module.ts* 文件中，我们必须在 `@NgModule` 定义中导入和添加每个提供者。下面的 `@NgModule` 定义中，在提供者列表中定义了 3 个插件：

```
@NgModule({
  declarations: [
  MyApp,
    TaskListPage
  ],
  imports: [
    IonicModule.forRoot(MyApp)
  ],
  bootstrap: [IonicApp],
  entryComponents: [
    MyApp,
    TaskListPage
  ],
  providers: [
    StatusBar,
    SplashScreen,
    Dialogs,
    { provide: ErrorHandler, useClass: IonicErrorHandler }
  ]
})
```

在 *tasklist.ts* 文件中，我们必须导入 `Dialogs` 模块。在其他 import 语句之后加入这句：

```
import { Dialogs } from '@ionic-native/dialogs';
```

在我们的构造器函数中，需要将 Dialogs 模块的一个引用以参数形式传递：

```
constructor(public navCtrl: NavController, public af: AngularFire,
public dialogs: Dialogs) {
```

然后，我们需要将代码替换为：

```
this.dialogs.prompt('Add a task', 'Ionic2Do', ['Ok', 'Cancel'], '').then(
  theResult => {
    if ((theResult.buttonIndex == 1) && (theResult.input1 !== '')) {
      this.tasks.push({ title: theResult.input1, status: 'open' });
    }
  }
)
```

### buttonIndex 值

你会奇怪，为什么我们会将 buttonIndex 和 1 进行比较，通常数组索引不是从 0 开始吗？其实 index 值 0 已经被用于指定对话框的解散按钮了，因此任何用户定义的按钮都不能使用这个索引。

如果你想稍微对我们的代码进行一点保护，我们可以在添加任务到列表之前对输入的字符串进行非空校验。图 7-9 展示了当 Add Item 按钮被单击后，我们的 App 的显示效果。

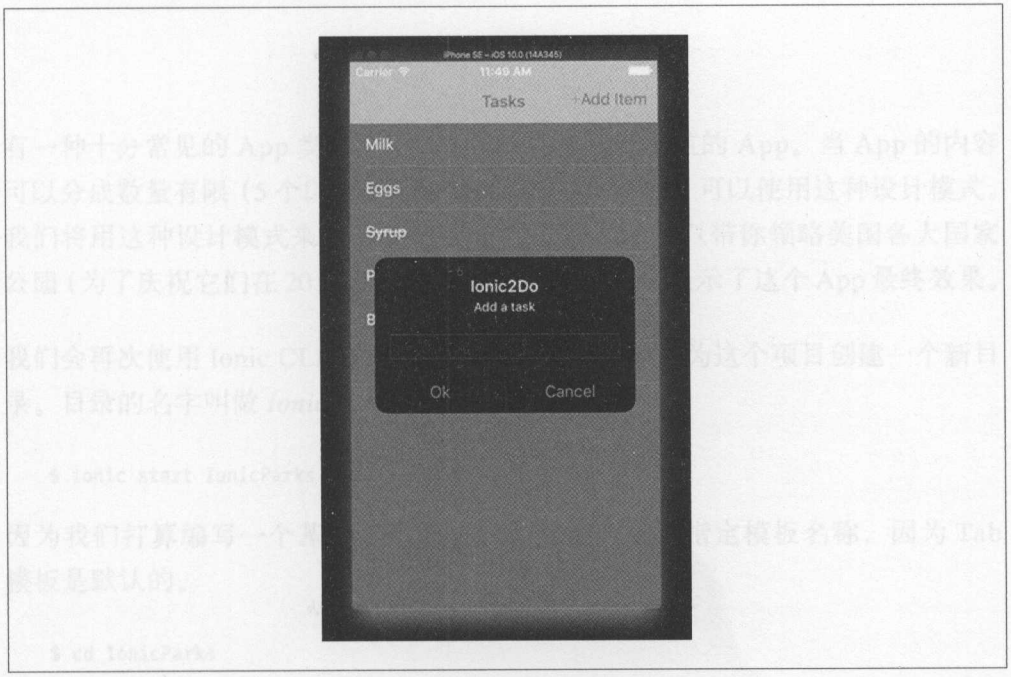



图 7-9：使用 Cordova 对话框插件来显示 Add Item 对话框

这样，一个基本的 to-do App 就已经做好了。我们学习了一些基本的 Ionic 组件和主题样式，调用了一个外部库，并整合了 Ionic Native 库。在下一章，我们会学习更多的 Ionic 组件和导航方法。

## 小结

通过这个 App，我们学习了基本的 Ionic 2 项目的构成。你学习了如何在 header 组件添加元素，使用列表，启用轻扫手势以显示按钮，以及使用 Ionicon 库。我们还加入了 FireBase 用于将数据保存到云端，并添加了 Ionic Native 组件来显示对话框以使我们的 App 看起来更像原生。

第 8 章

# 创建一个基于 Tab 的 App

有一种十分常见的 App 类型就是基于 Tab 方式进行导航的 App。当 App 的内容可以分成数量有限（5 个以下）的几个组（或者 tab）时，可以使用这种设计模式。我们将用这种设计模式来编写一个 App，这个 App 可以带你领略美国各大国家公园（为了庆祝它们在 2016 年的百周年庆典）。图 8-1 展示了这个 App 最终效果。

我们会再次使用 Ionic CLI 来创建 App 脚手架。首先，为这个项目创建一个新目录。目录的名字叫做 *IonicParks*：

```
$ ionic start IonicParks --v2
```

因为我们打算编写一个基于 Tab 的 App，我们不需要指定模板名称，因为 Tab 模板是默认的。

```
$ cd IonicParks
```

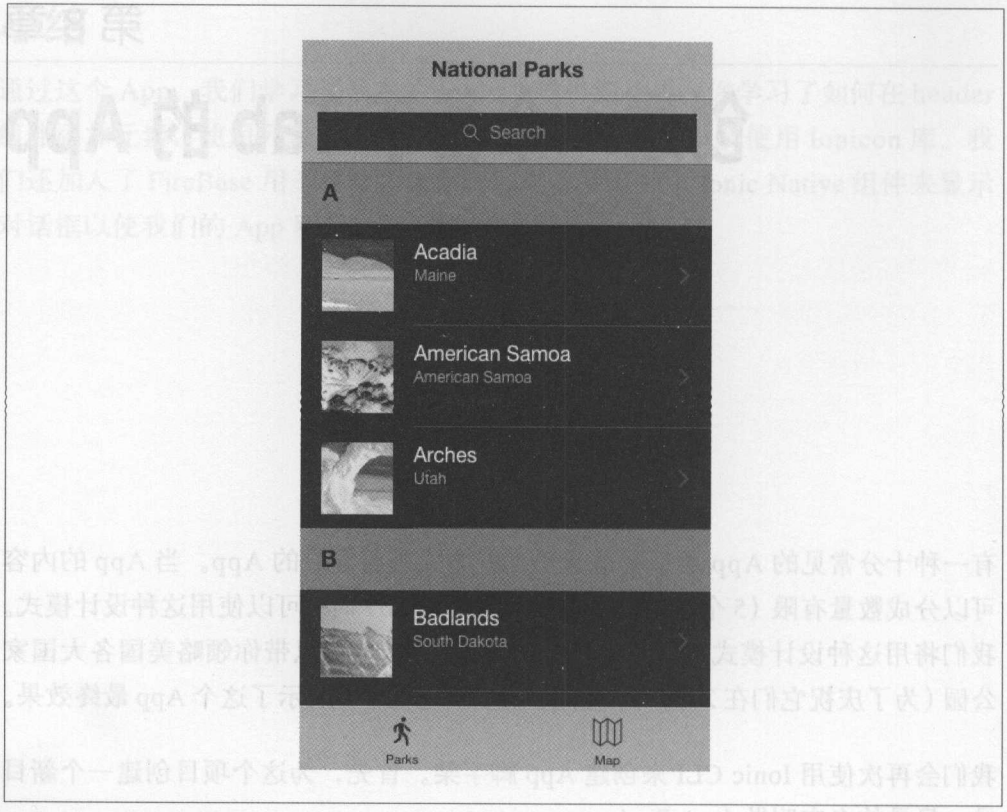



图 8-1：Ionic 国家公园 App

然后在继续后面的工作之前，先预览一下这个模板（见图 8-2）：

```
$ ionic serve
```

这里我们可以在 3 个 tab 之间来回切换（分别是 Home、About 和 Contact），并查看页面内容的改变。

看一眼 *app.module.ts* 文件，我们看到这里会导入 4 个页面，而不是仅仅 1 个页面。这 4 个页面分别构成了 3 个 Tab 页（HomePage、AboutPage 和 ContactPage），以及一个用于充当 App 容器的页面（TabsPage）。这些页面都包含和声明在 `entryComponents` 数组中。

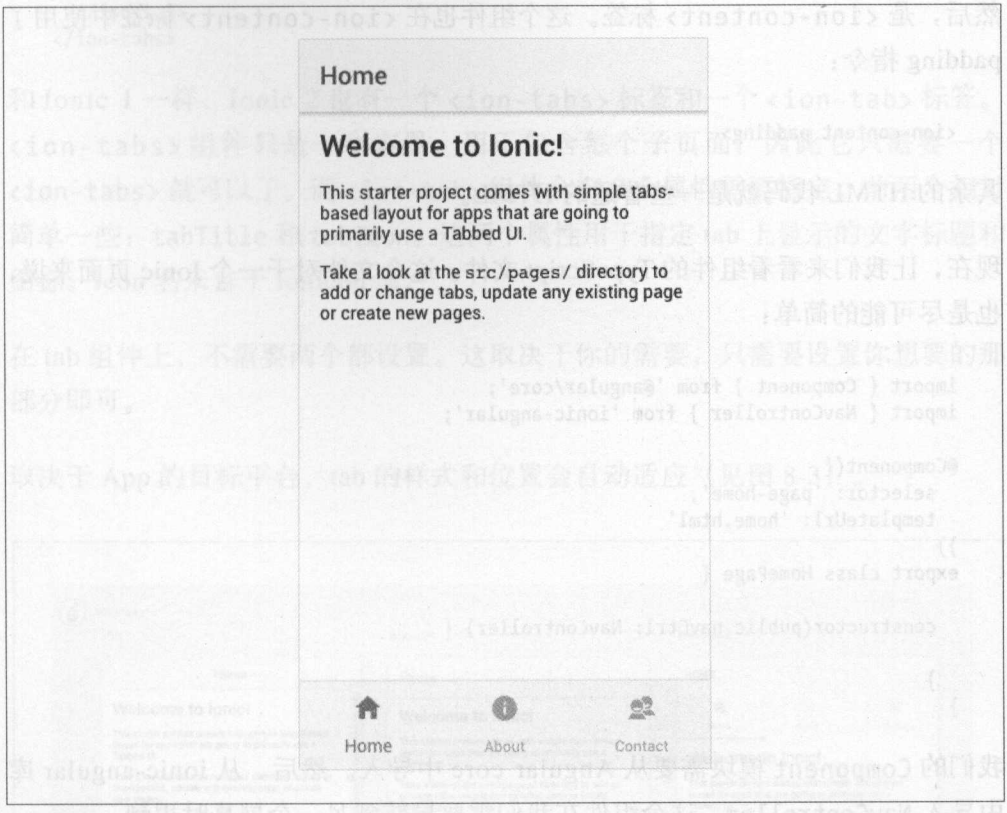



图 8-2：Ionic Tab 模板

然后打开 *app.component.ts* 文件，只有一个地方有所不同。现在 `rootPage` 是一个 TabsPage 而不是 table view。

再来看一眼 *pages* 目录。在这个目录中，我们会看到 4 个子目录：*about*、*contact*、*home* 和 *tabs*。打开 *home* 目录，里面放有和 home 页相关的 HTML、SCSS 和 TS 文件。

首先，HTML 定义了一个 `<ion-navbar>` 和 `<ion-title>` 组件：

```
<ion-header>
  <ion-navbar>
    <ion-title>Home</ion-title>
  </ion-navbar>
</ion-header>
```

然后，是 <ion-content> 标签。这个组件也在 <ion-content> 标签中使用了 padding 指令：

```
<ion-content padding>
```

其余的 HTML 代码就是一些普通的 HTML。

现在，让我们来看看组件的 TypeScript 文件。这个文件对于一个 Ionic 页面来说，也是尽可能的简单：

```
import { Component } from '@angular/core';
import { NavController } from 'ionic-angular';

@Component({
  selector: 'page-home',
  templateUrl: 'home.html'
})
export class HomePage {

  constructor(public navCtrl: NavController) {

  }
}
```

我们的 Component 模块需要从 Angular core 中导入。然后，从 ionic-angular 库中导入 NavController。这个组件在我们需要导航到另一个屏幕时用到。

在 @Component 修饰符中，我们将 templateURL 设置为 home.html，将 selector 设置为 page-home。

剩下的代码就是导出 HomePage 类，并将 NavController 传递给构造器函数。

其他两个 tab 页基本上和这个页面一致。每个页面都会导出一个组件，分别是 AboutPage 和 ContactPage。

现在来看一下 *tabs* 页自己。在 tabs 目录下，只有一个 *tabs.html* 和一个 *tabs.ts* 文件。先来看一下 HTML：

```
<ion-tabs>
  <ion-tab [root]="tab1Root" tabTitle="Home" tabIcon="home">↵
  </ion-tab>
  <ion-tab [root]="tab2Root" tabTitle="About" tabIcon="information-circle">↵
  </ion-tab>
  <ion-tab [root]="tab3Root" tabTitle="Contact" tabIcon="contacts">↵
```

```
  </ion-tab>
</ion-tabs>
```

和 Ionic 1 一样，Ionic 2 也有一个 `<ion-tabs>` 标签和一个 `<ion-tab>` 标签。`<ion-tabs>` 组件只是一个容器，用于包含每个子页面，因此它只需要一个 `<ion-tabs>` 就可以了。而 `<ion-tab>` 组件会有多个属性需要指定。前两个相对简单一些：`tabTitle` 和 `tabIcon`。这两个属性用于指定 tab 上显示的文字标题和图标。Icon 名来自于 IonIcon 库。

在 tab 组件上，不需要两个都设置。这取决于你的需要，只需要设置你想要的那部分即可。

取决于 App 的目标平台，tab 的样式和位置会自动适应（见图 8-3）。

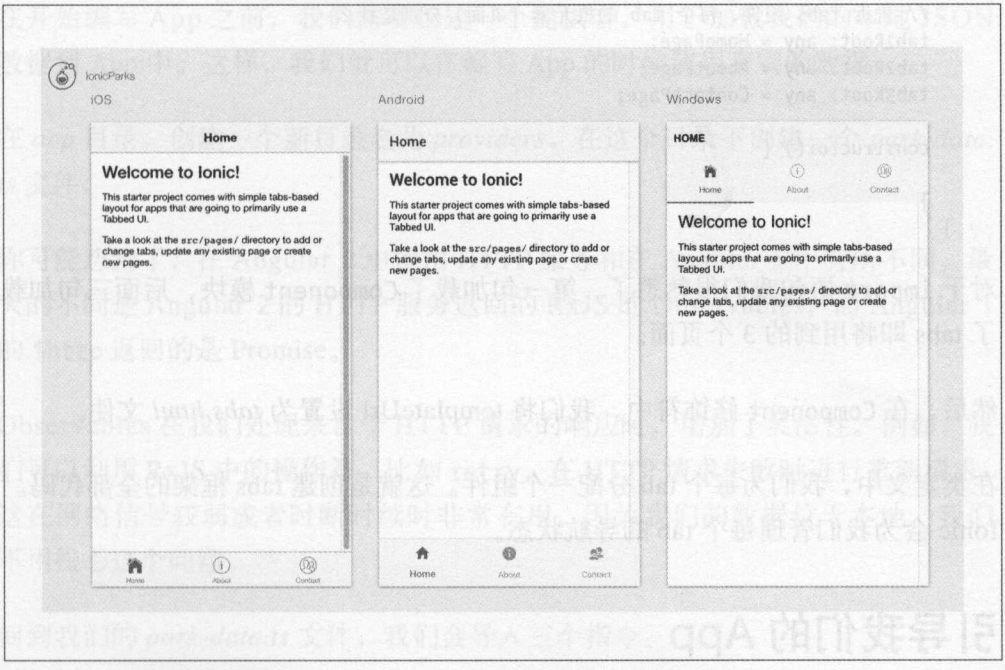



图 8-3：Tabs 组件基于平台类型进行显示

如果你想强制指定 tab 的位置，有两个办法，要么直接在组件上设置：

```
<ion-tabs tabsPlacement="top">
```

要么在 App 设置选项中进行设置。即在 *app.module.ts* 文件中进行设置：

```
IonicModule.forRoot(MyApp, {tabsPlacement: 'top'} )
```

要查看配置项的完整列表，请参考 Ionic 框架文档（*http://ionicframework.com/docs/api/config/Config/*）。

最后一个 `<ion-tab>` 属性是 `[root]` 绑定，这个属性用于指定用哪个组件来作为这个 tab 的根元素：

```
import { Component } from '@angular/core';
import { HomePage } from '../home/home';
import { AboutPage } from '../about/about';
import { ContactPage } from '../contact/contact';

@Component({
  templateUrl: 'tabs.html'
})
export class TabsPage {
  // 告诉 tabs 组件，每个 tab 的根元素（页面）分别是什么
  tab1Root: any = HomePage;
  tab2Root: any = AboutPage;
  tab3Root: any = ContactPage;

  constructor() {

  }
}
```

对于 `import` 语句我们很熟悉了。第一句加载了 `Component` 模块，后面三句加载了 tabs 即将用到的 3 个页面。

然后，在 `Component` 修饰符中，我们将 templateUrl 设置为 *tabs.html* 文件。

在类定义中，我们为每个 tab 分配一个组件。这就是创建 tabs 框架的全部代码。Ionic 会为我们管理每个 tab 的导航状态。

# 引导我们的 App

现在，我们对基于 tab 的 Ionic 2 App 的构成有了一定的了解，我们可以来修改我们的 App 了。这次你不需要搜索所有的文件和文件夹并将其重新命名，我们会用一种快速的方式。我已经初步修改好了这些模板文件。另外，我还会添加一个数据文件，用于保存各大国家公园的数据和图片。

首先，找到并删除 CLI 生成的 *IonicPark* 目录。然后从我的 GitHub 存储库创建我们的 App 脚手架：

```
$ ionic start Ionic2Parks https://github.com/chrisgriffith/Ionic2Parks --v2
```

当这个命令完成，再次切换工作目录：

```
$ cd IonicPark
```

如果你的目标平台是 Android，别忘记添加平台参数：

```
$ ionic platform add android
```

# 通过 HTTP 服务加载数据

在开始编写 App 之前，我们需要创建一个提供者，用于加载我们的本地 JSON 数据到 App 中。这样，我们就可以在编写 App 的时候使用真实的数据。

在 *app* 目录，创建一个新目录名为 *providers*，在这个目录下创建一个 *park-data.ts* 文件。

你可能想到了，在 Angular 2 中使用 HTTP 服务和在 Angular 1 中有所不同。最大的不同是 Angular 2 的 HTTP 服务返回的 RxJS 的 Observables，而 Angular 1 的 `$http` 返回的是 Promise。

Observables 在我们处理来自于 HTTP 请求的响应时，增加了灵活性。例如，我们可以利用 RxJS 中的操作符，比如 `retry`，在 HTTP 请求失败时进行重新请求。这在网络信号较弱或者时断时续时非常有用。因为我们的数据位于本地，我们不用操心这个问题。

回到我们的 *park-data.ts* 文件，我们会导入三个指令：

```
import { Injectable } from '@angular/core';
import { Http } from '@angular/http';
import 'rxjs/add/operator/map';
```

然后，我们就可以用 `@Injectable()` 修饰符了。Angular 2 文档提醒我们必须在 `@Injectable` 后面加上圆括号，否则 App 会无法编译。

我们来定义一个 ParkData 类作为提供者。现在添加一个变量 data，类型为 any，初始值设为 null。在构造器函数中，我们直接传递一个我们导入的 Http 对象作为参数，并将 Http 定义为公有 public。这个类的构造函数当前仍然为空：

```
@Injectable()
export class ParkData {
  data: any = null;

  constructor(public http: Http) {}

}
```

在类定义中，我们创建一个新方法 load。这个方法负责加载我们的 JSON 数据。实际上，我们要加入一些检查，确保数据在整个 App 生命周期只加载一次。方法的完整实现如下：

```
load() {
  if (this.data) {
    return Promise.resolve(this.data);
  }

  return new Promise(resolve => {
    this.http.get('assets/data/data.json')
      .map(res => res.json())
      .subscribe(data => {
        this.data = data;
        resolve(this.data);
      });
  });
}
```

因为我们硬编码了文件地址，我们不需要在调用这个方法时提供文件路径参数，因此用 load() 即可。

首先，这个方法判断 JSON 数据是否已经加载进 data 变量。如果已经加载，我们返回一个 Promise，对缓存的数据进行处理。我们必须将我们的数据转换成一个 Promise，这样才能将返回类型和 *park-list.ts* 中保持一致。在其他章节中，还会介绍 Observables 的使用。

我们来详细看一遍这段代码。如果数据未被加载，我们返回一个基本的 Promise 对象。然后，我们调用 HTTP 的 get 方法从 *assets/data/data.json* 地址读取 JSON 数据，然后再 resolve。在使用 Promise 的时候，需要指定它的 subscription（订

阅）。通过箭头函数 =>，结果被放到 res 变量。我们可以拿到这个字符串并利用 Angular 的内置函数 json() 转换为实际的对象。最终，我们用实际的数据 resolve 我们的 Promise。

这里，我们只是实现了提供者。现在，我们还要在我们的 App 中使用提供者。打开 *app.component.ts* 文件，在 import 语句中添加 `import { ParkData } from '../providers/park-data';`。这会加载提供者并将它赋给 ParkData。

接下来，我们要在 @Component 声明中指定一个 providers 数组：

```
@Component({
  templateUrl: 'app.html',
  providers: [ ParkData ]
})
```

现在来修改构造器函数，以加载我们的数据。我们会向构造器传入一个 ParkData 引用。在 platform.ready 代码块中，我们调用 parkData.load() 方法。这会告诉提供者去加载我们的数据文件：

```
constructor(platform: Platform, statusBar: StatusBar, splashScreen: SplashScreen,
            public parkData: ParkData) {
  platform.ready().then(() => {
    // Okay, so the platform is ready and our plugins are available.
    // Here you can do any higher level native things you might need.
    statusBar.styleDefault();
    splashScreen.hide();
  });
  parkData.load();
}
```

如果你想快速测试一下，看数据是否能够加载出来，将最后一句 parkData.load() 放到 console.log() 方法中。在 ZoneAwarePromise 的 __zone_symbol__value 中会创建出一个包含 59 个对象的数组。

### Zone 是什么？

一个 zone 是一种用于插入和跟踪异步操作的机制。因为我们的数据是以异步的方式加载的，必须使用 zone 来记录我们在请求时的原始上下文。

这是对每个公园数据进行格式化后的例子：

```
createDate: "October 1, 1890"
data: "Yosemite has towering cliffs, waterfalls, and sequoias in a diverse↵
      area of geology and hydrology. Half Dome and El Capitan rise from the↵
      central glacier-formed Yosemite Valley, as does Yosemite Falls, North↵
      America's tallest waterfall. Three Giant Sequoia groves and vast ↵
      wilderness are home to diverse wildlife."
distance: 0
id: 57
image: "yosemite.jpg"
lat: 37.83
long: -119.5
name: "Yosemite"
state: "California"
```

# 显示我们的数据

现在，App 已经通过我们的服务提供者读取到了数据，是时候正式将 59 条国家公园显示到列表中了。

首先，我们需要在 `ParkData` 服务提供者中添加一个方法，以返回我们加载好的数据。在 load 方法后，添加这个方法：

```
getParks() {
  return this.load().then(data => {
    return data;
  });
}
```

这个方法中，有两个地方需要注意。首先，它调用了 load 方法。load 会进行某些检查以确保数据一定存在。如果数据不存在，load 方法会为我们加载数据。因为我们的提供者使用了 Promise，构造一个链式调用是非常方便的。其次，因为使用了 Promise 模式，我们必须用 `.then` 模式来进行处理。这是从 Angular 1 迁移到 Angular 2 后必须注意的一些地方。

回到 *park-list.html*，在 `<ion-content>` 标签后面添加：

```
<ion-list>
  <ion-item *ngFor="let park of parks" ↵
            (click)="goParkDetails(park)" detail-push>
    <ion-thumbnail item-left>
      <img src="assets/img/thumbs/{{park.image}}">
    </ion-thumbnail>
    <h2>{{park.name}}</h2>
    <p>{{park.state}}</p>
  </ion-item>
</ion-list>
```

你还记得 Ionic2Do App 吗？我们需要定义一个 `<ion-list>` 组件，然后定义需要重复生成的 `<ion-item>`。重复的 item 是根据一个名为 `parks` 的数数组来产生的，这个数组待会定义。这个数组的每个元素都会被放到一个 `park` 变量中。在 `<ion-item>` 中加入了 click 事件处理器，它会调用一个名为 `goParkDetails` 的函数并将 park 变量作为参数传递进去。

如果你的 App 以 iOS 模式运行，会自动显示一个右箭头。在 Android 和 Windows 上，item 上不会添加这个右箭头。如果想显示这个默认不显示的右箭头，我们可以使用 `detail-push` 属性。否则如果我们不想显示右箭头，则用 `detail-none` 属性。我们还需要在 *variables.scss* 文件中的 `$colors` 变量定义之后添加一句代码以开启这个可视化状态：

```
$item-md-detail-push-show: true;
```

回到 *park-list.html* 文件，在 `<ion-item>` 中，我们插入了一个 `<ion-thumbnail>` 组件，并用 item-left 来设置它在这一行中的位置。img 标签非常简单，如果你使用了这个项目模板，应该有一个 img 目录，在 img 目录下有一个 thumbs 目录。Thumbs 目录放有每个公园的缩略图片。通过 Angular 的数据绑定，我们可以用 `src="assets/img/thumbs/ {{park.image}}"` 的方式动态地设置每张缩略图的 src 属性。然后，用 `<h2>` 和 `<p>` 标签显示公园名和州名，对应的数据和 park 对象进行绑定。

最后一件事情是删除 `<ion-content>` 中的 `padding` 属性。这会让列表占据整个 viewport 的宽度。修改完模板之后，我们可以转到组件代码的编写中来。

## 扩展 parklist.ts

我们需要干的第一件事情是将我们的服务提供者注入组件中：

```
import { ParkData } from '../../providers/park-data';
```

一开始，这个组件的类定义是空的。我们用下列代码替换它：

```
export class ParkListPage {
  parks: Array<Object> = []

  constructor(public navCtrl: NavController, public parkData: ParkData) {
    parkData.getParks().then(theResult => {
```

```
        this.parks = theResult;
      })
    }

    goParkDetails(theParkData) {
      console.log(theParkData);
    }
  }
```

我们定义了要在 HTML 模板中引用的 parks 变量：

```
parks: Array<Object> = [];
```

在组件类的构造函数参数中，我们定义了一个公共变量 parkData，类型为 ParkData。

然后，我们调用 parkData 的 getParks 方法。如果是以前，我们可能会写类似这样的代码来获取公园数据：

```
parks = parkData.getParks();
```

但自从我们开始使用 Promise 之后，我们实际上需要像这样来请求数据：

```
parkData.getParks().then(theResult => {
    this.parks = theResult;
    }
)
```

这就完成了对构造函数的修改。最后添加的代码是一个暂时未完成的函数，用于处理 <ion-item> 的单击事件。这个方法接收一个公园数据作为参数，然后简单地将数据写到控制台中。待会我们会讨论这个函数，现在先来用 $ ionic serve 命令来预览一下我们的工作（见图 8-4）。

当 App 启动之后，我们会在浏览器中看到一个国家公园列表，每个公园都列出了一个缩略图、标题和所在的州。如果单击某个 item，公园的数据会在 JavaScript 控制台中输出。现在我们已经写好了第一页，接下来将我们的视线转到公园的详情页。

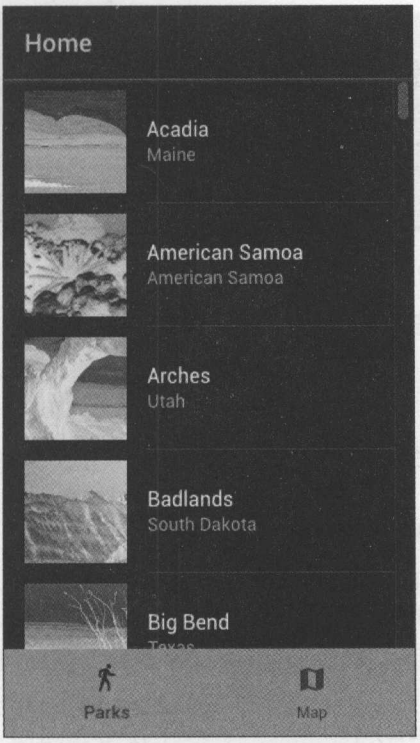



图 8-4：国家公园 App

## 生成新页面

在 Ionic 2 中，在 App 中添加新页面稍有点麻烦。幸运的是，在 CLI 中有一个生成页面的功能。要生成一个公园详情页的命令如下：

```
$ ionic g page parkDetails
```

CLI 会将小驼峰命名转换成烤肉串命名。生成器会自动在名字后面添加 `Page` 来作为类名。因此，如果你打开 *park-details.ts* 文件，你会看到类名是：

```
export class ParkDetailsPage { ...
```

我们还要在 *app.module.ts* 文件中加入对新组件的引用：

```
import { ParkDetailsPage } from '../pages/park-details/park-details';
```

然后将这个模块添加到 declarations 数组和 entryComponents 数组。

### Ionic 生成器

我们也可以用 Ionic CLI 生成器创建提供者，将 page 参数换成 provider 参数即可。实际上，我们之前编写的提供器就是用这种方式生成的。

现在，我们要在 *park-list.ts* 文件中编写能够导航到新页面的代码。我们必须从 Ionic core 中再导入一些模块。我们将第一句 import 语句修改为：

```
import { NavController, NavParams } from 'ionic-angular';
```

然后，需要导入要导航到的新页面：

```
import { ParkDetailsPage } from '../park-details/park-details';
```

将这些模块都注入我们的组件中，goParkDetails 函数现在可以跳转到公园详情页并传入公园信息给它了：

```
goParkDetails(theParkData) {
  this.navCtrl.push(ParkDetailsPage, { parkData: theParkData });
}
```

## 理解 Ionic 2 的导航模型

在 Ionic 1 时代，页面间的导航使用的是 AngularJS UI-Router。对于许多 App，这种导航方式都工作得非常好。但如果你的 App 具有一个复杂的导航模型，它就会出现问题了。例如，如果我们用 Ionic 1 来创建我们的 App，如果我们想访问公园详情页，我们必须使用两种不同的 URL，一种用于公园列表页，一种用于地图列表页。这些问题的存在迫使 Ionic 团队不得不重新建立全新的导航模型。

当前的导航模型基于 push/pop 系统。每个新的视图（或者页面）会被 push 到导航栈中，在这个导航栈中，一个视图会放在它前面的视图之上。当用户在栈中回退时，当前视图会从导航栈中移除（pop）（见图 8-5）。

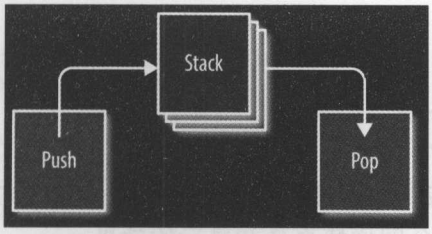



图 8-5：Ionic 2 的导航模型

新的导航模型更容易使用。

## 在页面中传递数据

在我们的 goParkDetails 函数中，它会从被单击的 item 中接收 parkData 参数。而使用 NavParams 模块，我们可以将这个数据传递给新页面的构造函数。

为了支持传入的数据，我们需要修改 *park-details.ts* 文件。在生成 Ionic 页面时，NavParams 模块已经从 ionic-angular 库导入了。然后，在类定义中，我们需要添加 parkInfo 变量，类型指定为 Object。

在构造函数中，导航参数被传入，并保存在 navParams 变量中：

```
import { Component } from '@angular/core';
import { NavController, NavParams } from 'ionic-angular';

@Component({
  selector: 'page-park-details',
  templateUrl: 'park-details.html'
})
export class ParkDetailsPage {
  parkInfo: Object;
  constructor(public navCtrl: NavController, public navParams: NavParams) {
    this.parkInfo = navParams.data.parkData;
    console.log(this.parkInfo);
  }
}
```

现在，我们仅仅将这个参数中保存的 parkData 输出到控制台中。我们所选定的公园的数据对象是保存在 navParams 的 data 方法中。保存文件，执行 $ ionic serve，单击某个 item，注意查看视图的改变以及控制台输出。

你可能注意到 Ionic 框架会处理页面导航之间的动画，同时自动在 header 上添加了一个返回按钮，以便用户能够在导航栈中回退。

## 修改公园详情页

现在我们已经能够跳转到公园详情页，让我们将重心转移到获取和显示动态数据上来。图 8-6 演示了公园详情页最初显示的效果。

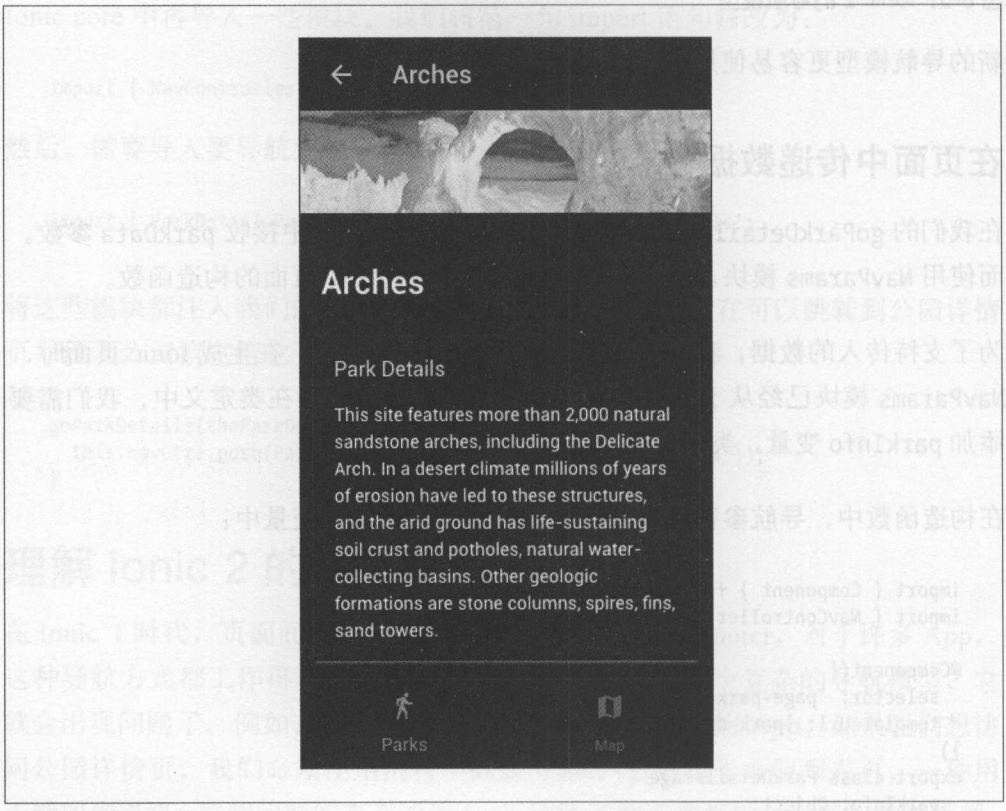



图 8-6：国家公园 - 详情页

生成的 HTML 页中已经存在一些基本的标签，但其中的大部分标签我们都会改到。首先，从页面头部移除 help 注释。将页面的 title 修改为 `{{parkInfo.name}}`：

```
<ion-header>
  <ion-navbar color="primary">
    <ion-title>{{parkInfo.name}}</ion-title>
```

```
  </ion-navbar>
</ion-header>
<ion-content>
    <img src="assets/img/headers/{{parkInfo.image}}">
    <h1 padding>{{parkInfo.name}}</h1>
    <ion-card>
        <ion-card-header>
            Park Details
        </ion-card-header>
        <ion-card-content>
            {{parkInfo.data}}
        </ion-card-content>
    </ion-card>
</ion-content>
```

在这里我们使用了一个新的组件 `<ion-card>`。根据文档描述，“Card 是一种用于显示重要的内容片段的方法，并迅速变成了 App 的一种基本的设计模式。”Ionic 的 card 组件是一个很灵活的容器，在 card 的内容中支持 header、footer，以及一堆别的组件。

准备好基本的公园详情页，可以用 `$ ionic serve` 预览一下。

## 添加 Google 地图

你可能猜到了，像国家公园这样的 App 肯定会显示某种类型的地图。不幸的是，Angular 2 没有正式提供一个 Google 地图模块。有一些第三方模块，但我们决定直接使用 Google 地图。这需要在 *index.html* 文件中引入这个库。因为 Google 地图禁止以离线方式缓存地图贴片，我们只能使用远程版本。

```
<!-- Google Maps -->
<script src="http://maps.google.com/maps/api/js"></script>

<!-- cordova.js required for cordova apps -->
<script src="cordova.js"></script>

<!-- The polyfills js is generated during the build process -->
<script src="build/polyfills.js"></script>

<!-- The bundle js is generated during the build process -->
<script src="build/main.js"></script>
```

在开发 API 时，我们可以不管 API key，但生产环境下还是需要使用一个 API key 的。你可以从 Google 开发者网站（*https://developers.google.com/maps/signup*）获得 API key。获得 API key 之后，修改 `<script>` 标签中 `src` 的 query 字符串。

## 添加额外的类型

因为我们在 App 中添加了第三方代码库，那么最好让代码提示和强类型也支持这个库的代码。我们可以扩展我们的 TypeScript 定义来实现这点。命令如下：

```
$ npm install @types/google-maps --save-dev --save-exact
```

## 添加内容安全策略

所谓的内容安全策略（Content Security Policy，CSP）是一个安全层，用于减少某些类型的恶意攻击，包括跨站脚本（XSS）。记住，我们的混合 App 仍然和 Web App 一样受到同样的限制。因此，我们也需要以同样方式来防护我们的 App。

在 *Index.html* 文件中，我们需要添加一个 CSP：

```
<meta http-equiv="Content-Security-Policy" content="default-src * gap://ready;↵
img-src * 'self' data:; font-src * 'self' data:; script-src 'self'↵
'unsafe-inline' 'unsafe-eval' *; style-src 'self' 'unsafe-inline'↵
*">
```

因为 Google 地图通过数据 URI 方法传输地图贴片，我们的 CSP 必须允许这种类型的通讯。此外，为了能够使用 Ionicons，还需要添加对 `font-src` 的支持。这个标签要放在 `<head>` 标签内。

## 修改 CSS 以支持 Google 地图

在我们的库能够加载和关联数据之后，我们将视线放到地图页面上。在 *park-map.html* 中，我们需要为地图添加一个容器以便显示地图：

```
<ion-content>
  <div id="map_canvas"></div>
</ion-content>
```

我们需要指定它的 CSS 样式的 `id` 或者 `class`，以便应用某些 CSS 样式。因为地图瓦片是动态加载的，第一次渲染时，我们的 `div` 不需要 width 或 height。这样，哪怕是地图瓦片已经加载了，这个容器的 width 和 height 也不会刷新。为了解决这个问题，我们必须定义 `div` 的 width 和 height。在 *park-map.scss* 文件中，添加代码：

```
#map_canvas {
    width: 100%;
    height: 100%;
}
```

这会给容器分配一个初始的宽高，我们的地图就可以看到了。

## 渲染 Google 地图

我们将在三个地方进行修改。首先是让 Google 地图开始显示，然后是在地图上添加每个国家公园的大头钉，最后是当单击大头钉使显示这个公园的详情页。打开 *park-map.ts* 文件。

我们需要从 `ionic-angular` 中导入 `Platform` 模块：

```
import { Platform, NavController } from 'ionic-angular';
```

我们将使用 `Platform` 模块来确保在 Google 地图创建之前所有的准备工作都就绪。

在类定义中，我们将定义一个 `map` 变量用于表示一个 Google 地图。这个变量会保存一个我们对 Google 地图的引用：

```
export class ParkMapPage {
    map: google.maps.Map;
```

然后，修改构造函数：

```
constructor( public nav: NavController, public platform: Platform) {
  this.map = null;
  this.platform.ready().then(() => {
    this.initializeMap();
  });
}
```

我们首先确保我们 `Platform` 模块可用，然后调用 platform 的 ready 方法返回一个 Promise。当平台的 ready 事件触发，我们以箭头函数的方式调用我们的 `initializeMap` 函数：

```
initializeMap() {
  let minZoomLevel = 3;
```

```
    this.map = new google.maps.Map(document.getElementById('map_canvas'), {
      zoom: minZoomLevel,
      center: new google.maps.LatLng(39.833, -98.583),
      mapTypeControl: false,
      streetViewControl: false,
      mapTypeId: google.maps.MapTypeId.ROADMAP
    });
  }
```

这个函数创建了一个新的地图对象，并将 id 为 map_canvas 的 div 对象赋给它。我们还指定了几个地图参数。这些参数包括缩放级别、地图中心（按照我们的要求，设置为美洲大陆的中心），以及各种地图控件和地图的风格。最后的对象方法是一个自定义方法，用于保存公园信息，这个方法会在本章后面的内容用到。

如果运行 $ ionic serve，我们会看到在 Map 标签页中显示一张地图，如图 8-7 所示。

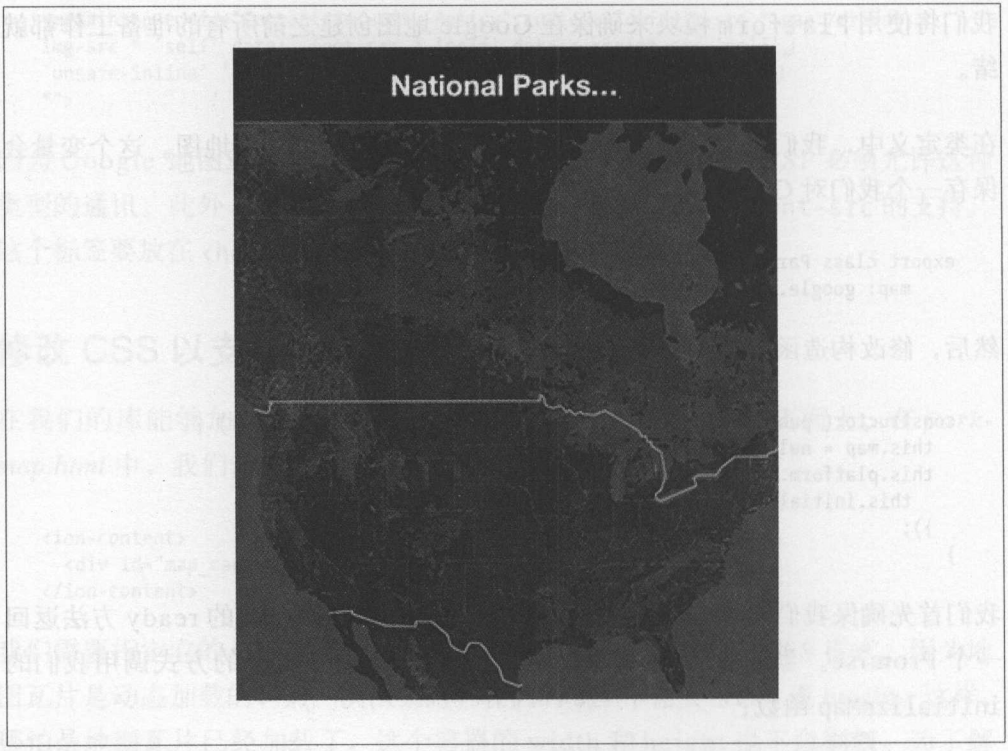



图 8-7：在我们的 Ionic App 中显示 Google 地图

## 添加大头钉

现在我们的 App 中有一个 Google 地图了，可以开始下一个任务：为每个国家公园添加一个大头钉到地图上。首先，我们需要向组件中导入 ParkData 服务：

```
import { ParkData } from '../../providers/park-data';
```

然后，我们要添加一个数组用于保存公园数据，确保这个类可以使用 parkData：

```
export class ParkMapPage {
  parks: Array<Park> = [];
  map: google.maps.Map;
  constructor(
    public nav: NavController,
    public platform: Platform,
    public parkData: ParkData
    ){
```

然我们可以将 parks 的类型设置为 any，但我们仍然还是将它的类型设置为符合公园数据结构的类型。因此，我们还要定义一个接口。在 *app* 目录下创建一个目录 *interfaces*。在新目录下面，新建一个文件 *park.ts*。这个文件用于描述一个简单的 Park 接口，代码如下：

```
export interface Park {
    id: number;
    name: string;
    createDate: string;
    lat: number;
    long: number;
    distance: number;
    image: string;
    state: string;
    data: string;
}
```

这个接口用于告诉编译器，Park 类型的数据将包含哪些元素和数据类型。

回到 *park-map.ts* 文件。我们需要导入这个接口文件：

```
import { Park } from '../../interfaces/park';
```

这会清除编辑器对 Park 数据类型的警告。

在 *park-list.ts* 文件中，也导入这个接口文件，然后将变量：

```
parks: Array<Object> = [];
```

修改为：

```
parks: Array<Park> = [];
```

在 `initializeMap` 函数中，我们需要添加显示大头钉的代码。

但我们不想使用 Google 的标准大头钉图片，我们想用国家公园服务中的箭头图标：

```
let image = 'assets/img/nps_arrowhead.png';
```

然后我们从 `parkData` 服务中获得公园数据。当 Promise 被 resolve，结果将通过 `parks` 数组来返回：

```
this.parkData.getParks().then(theResult => {
  this.parks = theResult;
  for (let thePark of this.parks) {
    let parkPos:google.maps.LatLng = ↵
        new google.maps.LatLng (thePark.lat, thePark.long);
    let parkMarker:google.maps.Marker = new google.maps.Marker();
    parkMarker.setPosition(parkPos);
    parkMarker.setMap( this.map);
    parkMarker.setIcon(image);
  }
})
```

我们在代码中遍历数组并生成大头钉。保存文件，如果 `ionic serve` 仍在运行，App 会重新刷新。选择 Map 标签 Tab，你会在地图上看到每个国家公园的图标。现在，这些图标还无法和用户交互。让我们来实现这个。

## 让大头钉可被单击

当用户单击到地图上的大头钉，我们想导航到对应的公园详情页。因此，我们需要从 Ionic 中导入导航模块和 `ParkDetailPage` 模块。修改后的 import 语句是这个样子：

```
import { Component } from '@angular/core';
import { Platform, NavController, NavParams } from 'ionic-angular';
import { Park } from '../../interfaces/park';
```

```
import { ParkData } from '../../providers/park-data';
import { ParkDetailsPage } from '../park-details/park-details';
```

在循环添加大头钉的 for 循环中，我们需要添加事件监听器以便响应单击事件，并根据传入的大头钉的公园数据导航到对应的 ParkDetailPage 页。不幸的是，标准的 Google 地图标记是不会有这个信息的。为了解决这个问题，我们需要创建自己的地图标记以便存储我们的公园数据。

新建一个文件 *custom-marker.ts*，放在 *park-map* 目录。这个新类将继承 google.maps.Marker，添加一个新的属性 parkData。我们首先要导入 Park 接口。然后导出我们的新类 CustomMapMarker，一个 google.map.Marker 子类。然后，我们定义一个 parkData 变量，类型指定为 Park。在类的构造器中，我们传入实际的公园数据。关键的代码是 super()。这句代码调用父类的初始化函数：

```
import {Park} from '../../interfaces/park';

export class CustomMapMarker extends google.maps.Marker{
  parkData:Park
  constructor( theParkData:Park
  ){
    super();
    this.parkData = theParkData;
  }
}
```

保存文件，返回 *park-map.ts* 文件。你可能猜到我们需要导入这个新写的类了。完全正确：

```
import { CustomMapMarker } from './custom-marker';
```

现在，我们的 parkMarker 可以用 CustomMapMarker 类替换掉 google.maps.Marker。因此将这句：

```
let parkMarker:google.maps.Marker = new google.maps.Marker();
```

替换为：

```
let parkMarker:google.maps.Marker = new CustomMapMarker(thePark);
```

我们向这个对象中传递了公园数据，这就可以将公园数据保存在每个大头钉中了。

现在我们来为每个大头钉添加一个事件监听器。但我们怎样才能从每个大头钉中获取对应的 parkData 以便将它放到 navParam 中去？

我们将采取一种取巧的方式。因为我们没有为我们的 CustomMapMarker 定义接口，编译器无法得知我们自定义的属性。但是，我们可以用 any 数据类型来规避这个问题。因此如果我们只要创建一个局部变量 selectedMarker，将它指定为 any 类型，然后将 parkMarker 赋给它，我们就可以引用到 parkData 了。代码如下：

```
google.maps.event.addListener(parkMarker, 'click', () => {
  let selectedMarker:any = parkMarker;

  this.navCtrl.push(ParkDetailsPage, {
    parkData: selectedMarker.parkData
  });
});
```

公园列表页的代码与此类似。这是完整的 initializeMap 函数代码：

```
initializeMap() {
  let minZoomLevel:number = 3;

  this.map = new google.maps.Map(document.getElementById('map_canvas'),    {
    zoom: minZoomLevel,
    center: new google.maps.LatLng(39.833, -98.583),
    mapTypeControl: false,
    streetViewControl: false,
    mapTypeId: google.maps.MapTypeId.ROADMAP
  });

  let image:string = 'img/nps_arrowhead.png';

  this.parkData.getParks().then(theResult => {
    this.parks = theResult;

    for (let thePark of this.parks) {
        let parkPos:google.maps.LatLng =↵
            new google.maps.LatLng (thePark.lat, thePark.long);
        let parkMarker:google.maps.Marker = new CustomMapMarker(thePark);
        parkMarker.setPosition(parkPos);
        parkMarker.setMap( this.map);
        parkMarker.setIcon(image);

        google.maps.event.addListener(parkMarker, 'click', () => {
          let selectedMarker:any = parkMarker;
          this.navCtrl.push(ParkDetailsPage, {
            parkData: selectedMarker.parkData
          });
        });
```

```
      }
    })
  }
```

保存文件，我们可以单击大头钉来查看公园详情了（见图 8-8）。

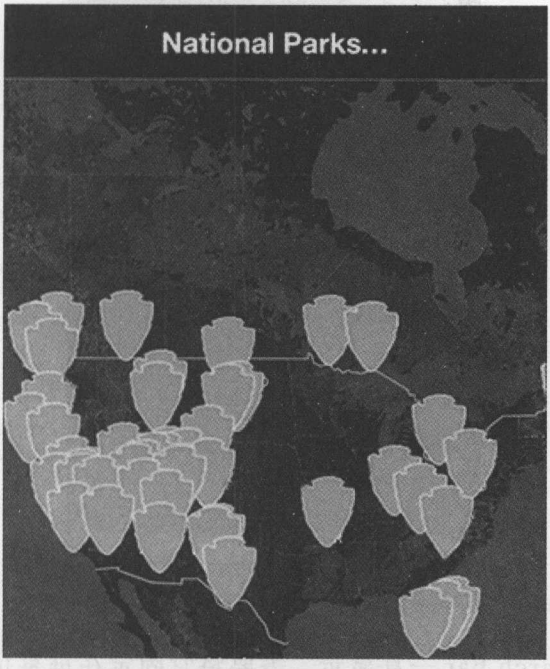



图 8-8：自定义 Google 地图的大头钉

## 添加查找功能

让我们给 App 增加一个新功能，在公园列表页面添加一个搜索栏。Ionic 的组件库中有一个 `<ion-searchbar>` 组件。这个搜索栏组件允许用户输入公园名称，公园列表会自动根据过滤后的结果进行刷新（见图 8-9）。

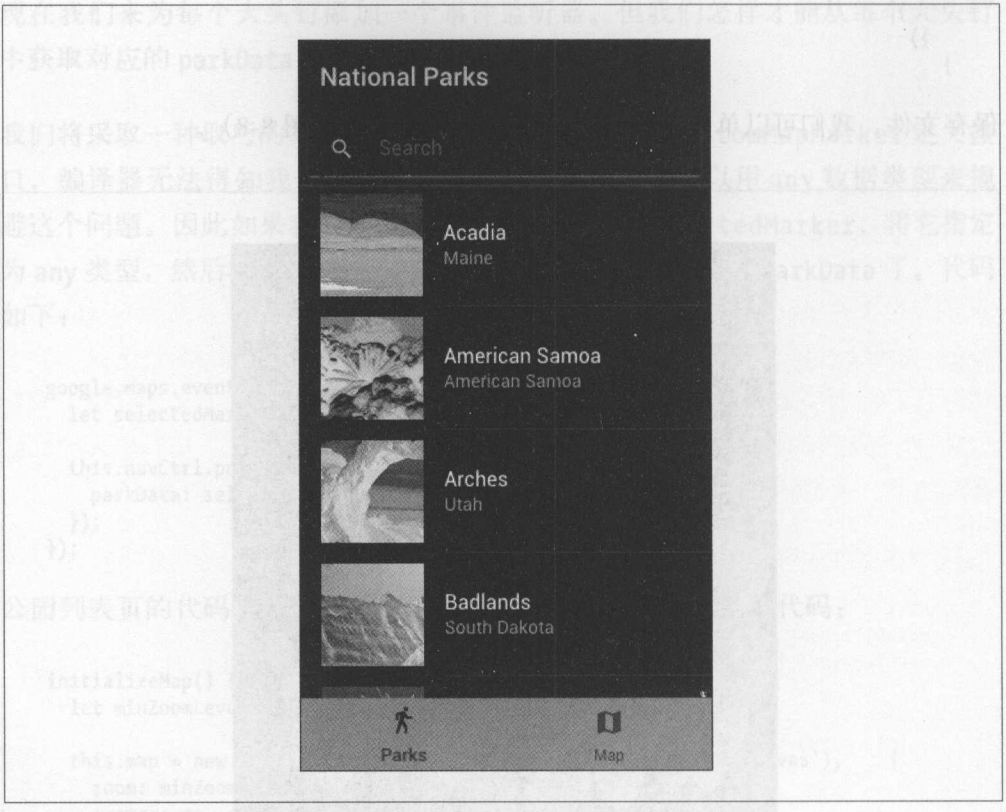



图 8-9：在我们的 Ionic App 中添加 Search bar 组件

因为我们想让 Search bar 一直显示，需要将它固定在屏幕顶部。我们可以用 `<ion-toolbar>` 组件来解决这个。这个组件需要放在 `<ion-navbar>` 后面。

我们需要为 `<ion-searchbar>` 定义一个模型并将它和查询字串进行绑定。同时，我们还要添加一个函数处理用户输入：

```
<ion-toolbar>
  <ion-searchbar [(ngModel)]="searchQuery" (ionInput)="getParks($event)" ↵
                 (ionClear)="resetList($event)">↵
  </ion-searchbar>
</ion-toolbar>
```

你是不是奇怪将 `ngModel` 用一个 `[( )]` 包裹起来是什么意思？这是 Angular 2 的双向数据绑定语法。方括号告诉 Angular 这是一个 getter 绑定，圆括号告诉 Angular 这是一个 setter 绑定。二者结合在一起，表示这是双向数据绑定。

在 *park-list.ts* 文件中，我们在类中定义 searchQuery 变量：

```
export class ParkListPage {
  parks: Array<Park> = []
  searchQuery: string = '';
```

同样在 *park-list.ts* 文件中，我们需要添加一个 getPartks 函数：

```
getParks(event) {
  // 将 items 重置回所有 items
  this.parkData.getParks().then(theResult => {
    this.parks = theResult;
  })

  // 将 queryString 设置为 searchbar 的文本
  let queryString = event.target.value;

  if (queryString !== undefined) {
    // 如果文本为空，不过滤 items
    if (queryString.trim() == '') {
      return;
    }

    this.parkData.getFilteredParks(queryString).then(theResult => {
      this.parks = theResult;
    })
  }
}
```

getParks 函数首先会使用原始的公园列表。如果你以前写过任何过滤函数，你就知道必须确保有一个未过滤的列表可以使用。

然后，我们从 search bar 获取查询字串，然后对它进行判断以确保它既不是 undefine 也不是空。

最后，我们会调用 parkData 提供者（*park-data.ts*）的新方法，并将返回结果赋给 parks 数组。这个新方法会基于查询字串进行过滤：

```
getFilteredParks(queryString) {
  return this.load().then(Parks => {
    let theFilteredParks: any = [];

    for (let thePark of Parks) {
      if (thePark.name.toLowerCase().indexOf(queryString.toLowerCase()) > -1) {
        theFilteredParks.push(thePark);
      }
    }
```

```
      return theFilteredParks;
    });
  }
```

我们首先会再次抓取公园数据，然后定义一个新的空数组用于将匹配的公园数据 push 进去。然后遍历每个 park，将公园名字和查询字串进行比较，将二者进行了字母的小写转换，再看二者是否匹配。如果匹配，将这个公园 push 到 `theFileteredParks` 数组。遍历完所有公园，返回 `theFilteredParks` 数组，公园列表会自动刷新。

搜索功能还没有完成。清除按钮还不能使用。虽然我们已经将 `ionClear` 事件绑定到 `resetlist` 函数，但这个函数还没写呢。这个函数其实很简单：我们只需要 `parks` 数组重置回完整的公园列表：

```
resetList(event) {
  // 重置回所有公园列表
  this.parkData.getParks().then(theResult => {
    this.parks = theResult;
  })
}
```

这样，我们的 App 的 search bar 的功能就完成了。

## 设置 App 的样式

我们已经拥有了一个可以运行的 App，我们还可以给它添加一些颜色使它焕然一新。关于 Ionic App 的样式化，这再简单不过了。在 Web App 或 Web 页面中也使用了同样的样式化技术。

Ionic 使用了 Sass 即 Syntactically Awesome Style Sheets 来作为 CSS 的预处理器。如果你从来没用过 CSS 预处理器，那么我们使用 CSS 预处理器的主要原因是它能将你的 CSS 中能够重用的部分进行抽离。例如，你可以定义一个变量，用于作为 App 的主要色，然后通过预处理在整个 CSS 中使用它。因此，一旦你需要修改 App 的主要色，你只需要在一个地方修改它。

Ionic 将它的 Sass 文件分成两个部分：*src/app/app.scss* 用于全局的、和 App 相关的样式，而 *src/theme/variables.scss* 用于预定义主题的颜色。

第一个要改变的样式是将 header 的颜色修改为森林绿。这可以有不止一种做法来实现这个：我们可以直接修改指定组件的样式，或者修改一个内置的主题。对于 header 组件，我们选择后一种方案。

在 *variables.scss* 文件中，将 primary color 的十六进制颜色值修改为 #5a712d。因为我们还没有在 header 或 tabs 组件上使用主题，所以我们必须先应用主题。修改三个页面的每一个 HTML 文件，将 `<ion-navbar>` 修改为 `<ion-navbar color="primary">`。

在 *tabs.html* 中，将 `<ion-tab>` 修改为 `<ion-tabs color="primary">`。保存所有文件，运行 `$ ionic serve`。header 和 Tabs 现在都会以森林绿作为背景色。

接下来修改各个内容元素的样式。让我们将 `<ion-content>` 正常的背景色修改为淡褐色。在 *app.css* 文件中，添加下列 CSS：

```
ion-content {background-color: #f4efdd;}
ion-card-header {background-color: #cfcbbb; font-weight: bold;}
ion-card-content{margin-top: 1em;}
ion-item-divider.item {background-color: #ab903c; color: #fff;
    font-size: 1.8rem; font-weight: bold !important;}
.item {background-color: #f4efdd;}
.toolbar-background {background-color: #706d61;}
.searchbar-input-container {background-color: #fff;}
```

你应该看到了，这些 CSS 是一种混合的样式，既有对 Ionic 组件的设置，也有对某种 CSS 类的设置。通过对 *app.scss* 的修改，整个 App 的样式都会被修改。如果你设置指定页面或组件，你只需要在对应的 *.scss* 文件中修改。让我们来试试看。

在公园详情页，公园的名称距离头部的图片有些偏低，同时我们还需要确保头部图片能够占据页面宽度：

```
page-park-details {
  h1{margin-top: 0;}
  img{width: 100%;}
}
```

现在，公园详情页的 `<h1>` 标签的上边距被设置为 0，而其他 `<h1>` 标签仍然使用原有的样式（见图 8-10）。


图 8-10：样式修改后的国家公园 App

Ionic 框架中其实已经包含了大量你可以覆盖的变量（*http://ionicframework.com/docs/theming/overriding-ionic-variables/*）。

如果你无法应用某个样式，记住它就是一个 CSS 而已，使用浏览器的 Web 检查器能够帮助你定位到具体的目标或者找出关联的问题。

## 虚拟滚动

混合 App 最大的一个性能问题就是在某个页面上存在了大量 DOM 元素。当滚动内容列表时，常常会出现问题。大部分用户会对混合 App 的滚动性能诟病不已。

认识到这个问题后，Ionic 团队花了大量工作创建出了一种能够几乎能够比拟原生的滚动体验。

我们的 App 只有 59 个公园，每一行所包含的内容也非常简单。如果将我们的 App 扩大到包含所有的国家名胜古迹，我们的列表可能会超过 400 行。这时，我们就会发现滚动的时候会出现一些卡顿和闪烁。

为了解决这个问题提，Ionic 框架引入了几个特殊指令，统一称作虚拟滚动。

### 什么是 Collection Repeat?

如果你用过 Ionic 1.x，你可能知道 Collection Repeat。它是一种专用于解决大数据滚动问题的机制。现在它已经被虚拟滚动所替代。

现在不再为列表中的每个 item 都创建 DOM 元素，而只有集合中的一小个子集（刚好填满一个 viewport）会被渲染，当用户滚动时进行重用即可。虚拟滚动负责在底层为你完成这一切。

在标准列表和使用了虚拟滚动的列表之间，存在着一些区别。首先，`<ion-list>` 现在用 `[virtualScroll]` 绑定到我们的数据。赋给 `virtualScroll` 属性的数据必须是一个数组。其次，`<ion-item>` 现在有一个名为 `*virtualItem` 的属性，这个属性引用了传递给模板的单个 item。

来看一下修改后的公园列表代码，你会更清楚一些：

```
<ion-list [virtualScroll]="parks">
  <ion-item *virtualItem="let park" (click)="goParkDetails(park)" detail-push>
      <ion-thumbnail item-left>
        <img src="assets/img/thumbs/{{park.image}}">
      </ion-thumbnail>
      <h2>{{park.name}}</h2>
      <p>{{park.state}}</p>
  </ion-item>
</ion-list>
```

除了将 `*ngFor="let park of parks"` 替换为两个虚拟滚动属性，其他代码仍然是一样的。但是，还有一个地方能够改进列表的滚动性能，那就是将 `<img>` 标签替换为 `<ion-img>` 标签：

```
<ion-img src="assets/img/thumbs/{{park.image}}"></ion-img>
```

这个标签专门用于虚拟滚动系统。<ion-img>标签负责处理 HTTP 请求和图片的渲染。另外，它还包含一个可以自己定义的占位图片，在图片加载完成后之前可以显示占位图片。在快速滚动时，<ion-img>并不会进行任何图片加载请求，只有在滚动结束后加载处于可见状态的图片。

关于性能的改善，你还可以采用以下策略：

- 当图片加载时，不要修改图片的大小。
- 提供大致的宽和高，方便虚拟滚动更好地算出 cell 的高度。
- 避免改变数据集，因为这会导致整个虚拟滚动被清空，这种操作的代价十分昂贵。

## 定制表格 header

虚拟滚动系统也支持动态添加 header 和 footer。以我们的 App 为例，我们可以让列表在每隔 20 行的时候添加一个 header：

```
<ion-list [virtualScroll]="items" [headerFn]="customHeaderFn">

  <ion-item-divider *virtualHeader="let header">
    {{ header }}
  </ion-item-divider>

  <ion-item *virtualItem="let item">
    Item: {{ item }}
  </ion-item>

</ion-list>
```

对应的函数实现如下：

```
customHeaderFn(record, recordIndex, records) {
  if (recordIndex % 20 === 0) {
    return 'Header ' + recordIndex;
  }
  return null;
}
```

体现到我们的国家公园列表中，我们的<ion-list>将变成：

```
<ion-list [virtualScroll]="parks" [headerFn]="customHeaderFn">
  <ion-item-divider *virtualHeader="let header">
```

```
      {{ header }}
    </ion-item-divider>

    <ion-item *virtualItem="let park" (click)="goParkDetails(park)" detail-push>
      <ion-thumbnail item-left>
        <ion-img src="assets/img/thumbs/{{park.image}}"></ion-img>
      </ion-thumbnail>
      <h2>{{park.name}}</h2>
      <p>{{park.state}}</p>
        </ion-item>
  </ion-list>
```

在我们的 *park-list.ts* 文件中，我们可以实现一个函数，在我们的自定义 header 中插入公园的第一个字母：

```
customHeaderFn(record, recordIndex, records) {
  if ( recordIndex > 0) {
    if ( record.name.charAt(0) !== records[recordIndex-1].name.charAt(0)) {
      return record.name.charAt(0);
    } else {
      return null;
    }

  } else {
    return record.name.charAt(0);
  }
}
```

最后一点工作是为 ion-time-divider 提供样式。在 *park-list.scss* 中加入：

```
page-park-list {
  ion-item-divider {
    background-color: #ad8e40;
    font-weight: bold;
  }
}
```

这样，我们的 App 就变成了这个样子（见图 8-11）。

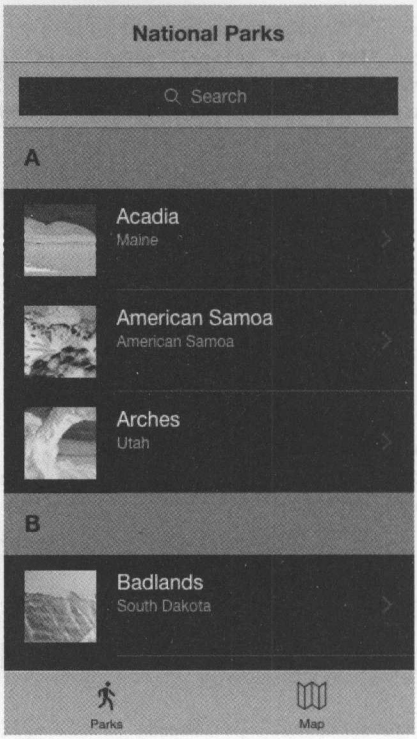



图 8-11：虚拟滚动 + 自定义动态 header

# 小结

通过这个 App，我们学习了如何使用基于 Tab 的设计模式，如何使用数据提供者、集成 Google 地图和设置基本的样式。如果你想继续改进这个 App，你可以在如下方面进行尝试：

1. 在公园详情页添加这个公园的地图。

2. 为每个公园添加一个图片轮播图。Ionic 有一个 `<ion-slides>` 组件，你可以用它来实现。

3. 如果你想真正挑战自己，可以计算到达每个公园的距离。在 *data.json* 文件中有一个 distance 属性。你可以利用 Geolocation 插件找出你当前的经纬度，然后用 Haversine 函数计算出距离。

第 9 章

# 构建一个天气应用

Ionic 框架还提供了另一个内置模板，即 side menu 模板。这种设计模板在过去几年来越发流行。比起拥有固定数目的 tab bar（常常会固定在屏幕上占据宝贵的屏幕空间）来说，这种用户界面将大量导航选项放到了一个面板里面，这个面板不会占据屏幕空间，除非用户单击某种形式的菜单按钮。通常这个按钮会是一个三层横线构成的图标（即汉堡菜单）或者三个竖直排列着的点。现在，我先不说这种用户界面元素的优劣。我建议你自己花点时间来研究它，看是否可以用在你的项目中。总之，我们会在我们的 Ionic 天气应用中使用这种设计模式。

## 开始

和前面两个项目一样，先来生成我们的初始的项目。我们使用另一个 GitHub 存储库作为开始模板。这个基本的模板就是 side menu 模板，只不过在 assets 文件夹中包含了一些额外的内容。

```
$ ionic start Ionic2Weather https://github.com/chrisgriffith/Ionic2Weather --v2
```

当项目承建完毕，记得切换工作目录：

```
$ cd Ionic2Weather
```

如果你的目标平台是 Android，别忘了添加平台：

```
$ ionic platform add android
```

然后在浏览器中看一下这个模板：

```
$ ionic serve
```

图 9-1 展示了你在浏览器中会看到的内容。

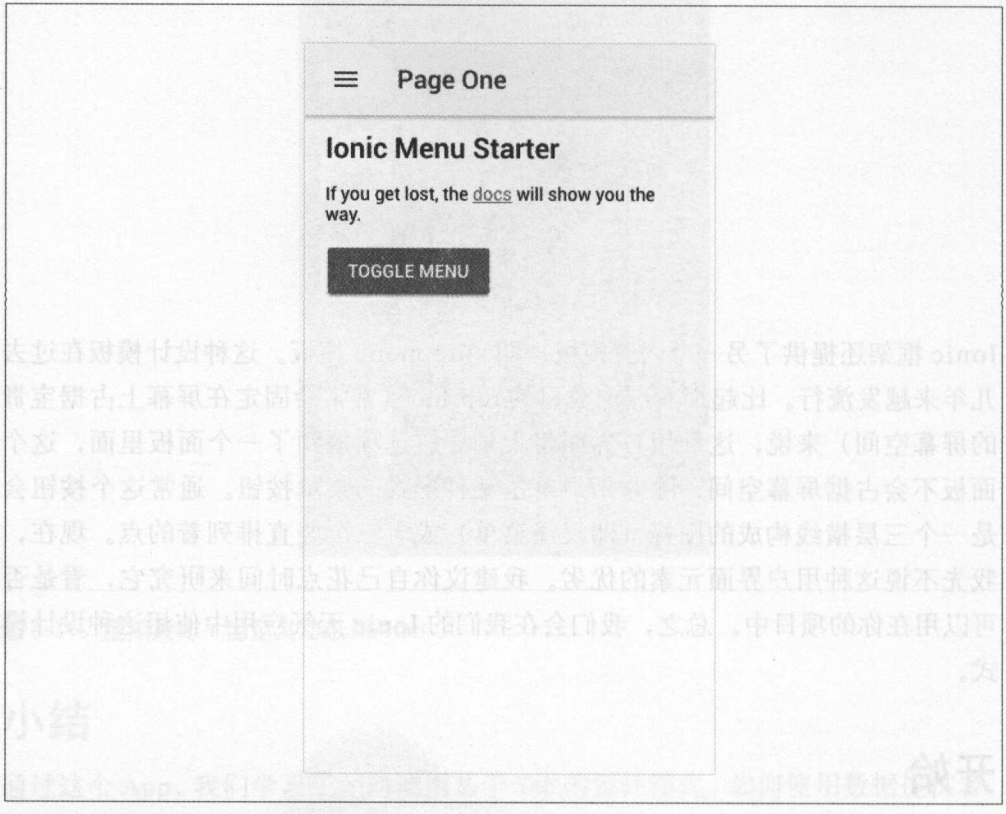



图 9-1：初始的 side menu 模板

这个模板非常简单。主界面中可以用两种方法调出侧滑菜单：navbar 上的菜单按钮或者内容区域的按钮。当你单击这两个按钮，菜单会以动画方式滑入屏幕，原有内容会以半透明的方式遮在在下面。真正动画取决于 App 的具体平台。当然，动画是可以自定义的。

侧滑菜单包含了两个部分：一个是它自己的 navbar，即一个写有 Menu 字样的文本标签，另一个是它的内容，即一个拥有两个条目的列表（Page One 和 Page Two）。如果你单击侧滑菜单以外的区域，Ionic 会自动隐藏侧滑菜单。如果你单击 Page One 或者 Page Two，App 主界面中的内容会随之改变，侧滑菜单消失。

## 探究侧滑菜单模板

side menu 模板比另外两个模板要复杂点，因此我们在动手编写天气应用之前先来看一下模板的基本构成。默认的 *index.html* 没有什么不同的，因此不需要看它。让我们先从 *app.html* 文件开始。这个文件位于 *src/app* 目录。这是我们 App 要显示的第一个 HTML。和另两个模板不同，这个模板的 App 组件引用了一个外部的 HTML 模板（*app.html*）而不是内联模板。和我们看到的一样，这个文件比 blank 模板和 tab 模板的要复杂，将它作为一个外部文件来引用显然要好一些。这个文件的内容如下：

```
<ion-menu [content]="content">
  <ion-header>
    <ion-toolbar>
      <ion-title>Menu</ion-title>
    </ion-toolbar>
  </ion-header>

  <ion-content>
    <ion-list>
      <button menuClose ion-item *ngFor="let p of pages" (click)="openPage(p)">
        {{p.title}}
      </button>
    </ion-list>
  </ion-content>

</ion-menu>

<!-- Disable swipe-to-go-back because it's poor UX to combine STGB
with side menus -->
<ion-nav [root]="rootPage" #content swipeBackEnabled="false"></ion-nav>
```

这和 Ionic 1 的侧滑菜单模板有明显的不同。最显著的一点是 `<ion-side-menu>` 和 `<ion-side-menu-content>` 标签没有了，而用 `<ion-menu>` 标签和新页面引用系统取代。

让我们再仔细瞄一眼 `<ion-menu>` 标签。这个标签用于创建需要在侧滑菜单中显示的内容。但在这个标签中有一个关键属性，`[content]= "content"`。我们将 ion-menu 的 content 属性设置为一个变量 content，而不是一个字符串。如果你找一下这个文件的最后一个标签，你会看到有一个 `#content`，它定义了一个本地变量。现在我们的 `<ion-menu>` 可以引用它并用它作为主要内容。我们在以后还会看到 ion-nav 的其他属性。

然后，我们用 <ion-toolbar> 标签来定义侧滑菜单的 header。这个组件是一个普通的 bar，用于放在 content 的上面或下面，就像 <ion-nav>，只不过没有导航控件而已：

```
<ion-toolbar>
  <ion-title>Menu</ion-title>
</ion-toolbar>
```

然后，在 <ion-content> 中定义侧滑菜单的内容：

```
<ion-content>
  <ion-list>
    <button menuClose ion-item *ngFor="let p of pages" (click)="openPage(p)">
      {{p.title}}
    </button>
  </ion-list>
</ion-content>
```

这段代码用一个 <ion-list> 遍历 pages 数组并创建一系列 button，button 的标题使用对象的 title 属性。同时还绑定了一个单击事件处理函数 openPage，以响应用户单击。

然后，回到 <ion-nav> 标签，仔细瞄一眼：

```
<ion-nav [root]="rootPage" #content swipeBackEnabled="false"></ion-nav>
```

root 属性被设置为 rootPage。rootPage 在 *app.component.ts* 文件中定义，当我们需要改变我们的主内容时，我们会更新这个属性。现在，让我们回到 *app.componet.ts* 文件，看一下这几部分是如何联系在一起的。

## 探索 app.component.ts 文件

和侧滑菜单的 HTML 模板一样复杂的就是 *app.component.ts* 文件。首先，我们一来就看见几个 import 语句：

```
import { Component, ViewChild } from '@angular/core';
import { Nav, Platform } from 'ionic-angular';
import { StatusBar, Splashscreen } from 'ionic-native';

import { Page1 } from '../pages/page1/page1';
import { Page2 } from '../pages/page2/page2';
```

除了导入 Component 模块，我们还要从 @angular/Core 中导入 ViewChild 组件。通过这个 import 语句我们的代码就可以用 @ViewChild 修饰符去引用子元素。底层有一种机制允许我们正确地引用我们的 nav 元素。

接下来两个 import 语句对你来说并不陌生。我们需要从 Ionic 导入 Nav 和 Platform 指令，并从 Ionic Navtive 导入 StatusBar。

最后两个 import 语句是示例中的两个页面模板。因为我们的 *app.component.ts* 负责处理侧滑菜单的导航，所以要引用任何可以被导航的页面。

我们的 @Component 修饰符看起来很熟悉。就像之前说过的，它引用了一个外部模板。

然后，我们定义了我们的 rootPage 变量，类型定义为 any，将它的初始值设置为 Page1 组件：

```
rootPage: any = Page1;
```

定义一个数组，用于保存侧滑菜单的页面列表，也就是它的内容：

```
pages: Array<{title: string, component: any}>
```

这个例子极好的展示了 TypeScript 如何通过指定元素类型来保证代码的安全。

构造函数对 App 进行初始化（对于 iOS 来说，当 platform 就绪时，这是一个对状态栏进行处理的时机），并将 pages 数组初始化为两个示例页面。

最后，我们定义一个 openPage 函数，用于当侧滑菜单按钮被单击时调用：

```
openPage(page) {
  // 将 content 跳回当前页面，
  // 在这种情况下我们不需要返回按钮
  this.nav.setRoot(page.component);
}
```

这个函数接收一个 page 对象作为参数，然后将 nav 控制器的 root 属性设置为这个 page。通过设置 nav 的 root，我们可以重设导航栈历史。同时，我们还通过这种方式让 Ionic 不会自动添加返回按钮。

## 侧滑菜单选项

你可能看到了，实际上我们并没有为侧滑菜单指定任何菜单项。

菜单既可以放在左边（默认）也可以放在右边。要改变侧滑菜单的位置，可以使用 side 属性：

```
<ion-menu side="right" [content]="content">...</ion-menu>
```

侧滑菜单支持三种显示风格：overlay、reveal 和 push。Ionic 框架会根据当前所运行的平台自动采用对应的风格。默认，对于材料设计和 Windows 的显示风格是 overlay，iOS 的默认显示风格则是 reveal。

如果你想改变显示风格，可以在 <ion-menu> 标签中这样做：

```
<ion-menu type="overlay" [content]="content">...</ion-menu>
```

同时，你还可以通过 App 的 config 对象以全局方式设置侧滑菜单的显示风格。也就是在 *app.module.ts* 文件中进行设置。以下代码将默认的显示风格设置为 push，对于材料设计（Android），则将 menuType 设置为 reveal：

```
imports: [
  IonicModule.forRoot(MyApp, { menuType: 'push',
  platforms: {
   md: {
     menuType: 'reveal',
   }
  }})
]
```

## 显示菜单

真正要显示我们的菜单，我们还需要在模板中使用 menuToggle 指令。我们先看一眼 *page1.html* 文件（*src/pages/page1*），你会看到有一个 <ion-navbar>HTML 块：

```
<ion-header>
  <ion-navbar>
    <button ion-button menuToggle>
      <ion-icon name="menu"></ion-icon>
    </button>
    <ion-title>Page One</ion-title>
  </ion-navbar>
</ion-header>
```

在 <button> 元素中，你会看到使用了一个 menuToggle 指令。这就是我们要显示的侧滑菜单时所要做的全部。这里没有为侧滑菜单按钮定义任何可视化元素。仅仅是在 <ion-icon> 中将 name 设置为 *menu* 来就行了。

事实上，如果你继续查看 *page1.html* 中的代码，你还会看到：

```
<button ion-button secondary menuToggle>Toggle Menu</button>
```

这个按钮也使用了一个 menuToggle 指令，这也允许这个按钮能够打开侧滑菜单。尽管 Ionic 框架会为我们解决一部分打开和关闭侧滑菜单的工作，但有时候我们仍然需要直接关闭侧滑菜单。我们可以用 menuClose 指令来实现这个功能：

```
<button ion-button menuClose="left">Close Side Menu</button>
```

这样我们就介绍完了大部分基本的侧滑菜单模板。让我们开始编写自己的 App 吧。

## 转换模板

在我们开始将模板转换为我们想要的之前，我们先生成两个在这个 App 中用到的页面，以及数据提供者：

```
$ ionic g page weather

$ ionic g page locations
```

接下来，添加两个提供者。第一个提供者是用于从 darksky.net（以前叫 Forecast.io）拉取在线天气数据的提供者：

```
$ ionic g provider WeatherService
```

执行完这个创建提供者的命令，我们需要创建第二个提供者。第二个提供者用于将一个对人友好的地名，比如圣迭戈，转换成 darksky.net 所需的经纬度。在本章稍后的内容中再讨论这两个提供者：

```
$ ionic g provider GeocodeService
```

现在，我们的基本页面和提供者就准备好了，让我们开始修改模板以便使用它们。用你的编辑器打开 *app.module.ts*。

在 import 语句中，我们可以删除 Page1 和 Page2 的引用，然后导入我们的两个新创建的页面：

```
import { WeatherPage } from '../pages/weather/weather';
import { LocationsPage } from '../pages/locations/locations';
```

同时，修改 declarations 数组和 entryComponents 数组，替换上我们的新页面。

在导入语句之后，我们还要导入两个提供者：

```
import { WeatherService } from '../providers/weather-service';
import { GeocodeService } from '../providers/geocode-service';
```

在 providers 数组中，我们添加进我们的两个提供者：

```
providers: [
  StatusBar,
  SplashScreen,
  Geolocation,
  { provide: ErrorHandler, useClass: IonicErrorHandler },
  WeatherService,
  GeocodeService
]
```

改好后，我们打开 *app.component.ts*，准备修改它。首先修改两个 import 语句，替换上我们的新页面：

```
import { WeatherPage } from '../pages/weather/weather';
import { LocationsPage } from '../pages/locations/locations';
```

仍然在这个文件中，我们会看到定义 rootPage 变量的那一句。我们需要将其中的 Page1 修改为我们的 WeatherPage。

```
rootPage: any = WeatherPage;
```

接下来需要修改 pages 数组。我们会在编写 App 的过程中继续充实这个数组，但目前，我们只需要进行最基 本的修改。

因为 App 是使用 TypeScript 来编写的，在修改模板时我们必须小心谨慎，我们应当保证每个变量的定义是正确的。例如，pages 变量被定义为一个 object 数组，每个 object 会包含一个 string 类型的 title 属性和一个 any 类型的 component 属性。同时我们想在菜单列表的每个 item 上显示一个图标，所以我们应当这样定义它：

```
pages: Array<{title: string, component: any, icon: string}>
```

让我们来修改 pages 数组，让它能够引用正确的 component，同时添加一个 icon 和 title。新的数组应该是：

```
this.pages = [
  { title: 'Edit Locations', component: LocationsPage, icon: 'create' },
  { title: 'Current Location', component: WeatherPage, icon: 'pin' }
];
```

稍后，我们会扩充这个数组，让它保存我们的位置和经纬度。

如果你运行 App，你只会看到天气页面，并无法跳转到侧滑菜单。我们要解决这个问题，需要修改一下 navbar。

在你的编辑器中打开 *weather.html*。

首先，我们需要在 `<ion-navbar>` 中添加侧滑菜单按钮：

```
<button ion-button menuToggle>
  <ion-icon name="menu"></ion-icon>
</button>
```

然后，将 `<ion-title>` 修改为 `Current Location`。当我们实现地理定位功能后，再回来修改这个标签，让它显示真正的位置：

```
<ion-title>Current Location</ion-title>
```

如果你现在保存文件，你会看到 menu 图标已经显示在左边，并且能够打开我们的侧滑菜单了。同样，我们也需要修改 *locations.html* 文件中的 header：

```
<ion-header>
  <ion-navbar>
    <button ion-button menuToggle>
      <ion-icon name="menu"></ion-icon>
    </button>
    <ion-title>Edit Locations</ion-title>
  </ion-navbar>
</ion-header>
```

添加完代码之后，这个页面页就能够显示侧滑菜单了。回到 *app.html* 文件，修改代码以显示我们的每个列表项的图片。我们只需要添加一个 `<ion-icon>` 标签并设置 name 属性为列表项的 icon 属性。我们还需要在图标右边添加一点空白

间距，以便让文本不会紧挨着图标。最简单的办法是在 `</ion-icon>` 结束标记和 `{{p.title}}` 之间添加一个空格：

```
<ion-list>
  <button menuClose ion-item *ngFor="let p of pages" (click)="openPage(p)">
    <ion-icon name="{{p.icon}}"></ion-icon> {{p.title}}
  </button>
</ion-list>
```

如果想更好地控制图标和位置之间的间隔，我们可以用 CSS。

另一个需要调整的地方是将 `<ion-title>` 从默认的 Menu 修改成：

```
<ion-title>Ionic Weather</ion-title>
```

我们最后需要改的地方是添加一个链接，表明我们是从哪里获得天气数据的（你真的看完了协议条款了吗）。在 `<ion-list>` 之后，添加这个标签：

```
<p><a href="https://darksky.net/poweredby/">Powered by Dark Sky</a></p>
```

保存文件，我们的侧滑菜单能够显示出我们的图标并能够导航到我们自己的页面。

## 模拟天气提供者

我们的下一步是获取天气数据，以便在我们的 App 中使用它们。我们将分成两个阶段来实现。首先我们会加载静态的天气数据。这会方便我们进行初步的屏幕布局。然后我们将 App 切换到真实的数据源。

在 asset 目录下有一个静态的数据源，我们可以使用它。这个文件是从 darksky.net 的天气数据中截取的。

现在，让我们修改天气提供者的代码，添加一个方法来加载数据。打开 *weather-service.ts* 文件，将内容修改为：

```
import { Injectable } from '@angular/core';
import { Http } from '@angular/http';
import 'rxjs/add/operator/map';

@Injectable()
export class WeatherService {
  data: any = null;
```

```
  constructor(public http: Http) {
    console.log('Hello WeatherService Provider');
  }

  load() {
    if (this.data) {
      return Promise.resolve(this.data);
    }

    return new Promise(resolve => {
      this.http.get('assets/data/data.json')
        .map(res => res.json())
        .subscribe(data => {
          this.data = data;
          resolve(this.data);
        });
    });
  }
}
```

在 load 方法之后，添加这个函数：

```
getWeather() {
  return this.load().then(data => {
    return data;
  });
}
```

尽管这个方法只是调用了 load 函数，我还是宁愿将它写成一个单独的方法。这会在关键的时候增加一些灵活性。

现在，我们必须让 App 告诉这个提供者，让它为我们加载数据。回到 *app.compoent.ts* 文件，我们需要为调用提供者进行一些修改。

首先，我们必须将提供者导入这个组件：

```
import { WeatherService } from '../providers/weather-service';
```

在构造函数中，我们必须将我们的 WeatherData 提供者传递进去：

```
constructor(publice platform: Platform,
public weatherService: WeatherService) {...}
```

现在，在 initializeApp 函数中，就可以调用这个服务并加载我们的数据了：

```
initializeApp() {
    this.platform.ready().then(() => {
      // Okay, so the platform is ready and our plugins are available.
      // Here you can do any higher level native things you might need.
      this.statusBar.styleDefault();
    this.splashScreen.hide();
    this.weatherData.load();
  });
}
```

准备好数据之后，我们就可以来显示它了。

## 显示天气数据

在显示我们的模拟天气数据之前，我们必须在页面上下文中使用它。打开 *weather.ts* 文件。首先我们需要导入 WeatherService 提供者：

```
import { WeatherService } from '../../providers/weather-service';
```

然后，我们需要声明三个变量，用于以 class 定义的形式保存我们的天气数据。我们将这些变量声明为 any，以使它不报 TypeScript 错误：

```
theWeather: any = {};
currentData: any = {};
daily: any = {};
```

这里定义 currentData 和 daily 变量的原因是因为 darksky.net 的 JSON 数据结构是嵌套的。为了不用在我们的模板中进行递归，我们通过定义 currentData 和 daily 变量来进行引用。

然后，在构造函数中传递一个对 WeatherService 的引用：

```
constructor(public navCtrl: NavController, public navParams: NavParams,
public weatherService: WeatherService) { }
```

现在，在构造函数中，我们可以调用天气数据提供者的 getWeather 方法了：

```
this.weatherService.getWeather().then(theResult => {
  this.theWeather = theResult;
  this.currentData = this.theWeather.currently;
  this.daily = this.theWeather.daily;
});
```

数据已经加载好了，让我们回到 HTML 模板上来。再次打开 *weather.html*，我们将修改这个模板，以便显示我们的天气数据。

要显示数据，我们准备使用 Ionic Grid 组件。在 Ionic 文档中是这样描述这个组件的：

> Ionic 的网格系统基于弹性盒子模型，Ionic 所支持的设备都支持这个 CSS 特性。这个网格由三部分构成：网格、行、列。列填充行，当添加新列时自动调整大小。

首先，我们要添加一个 `<ion-grid>` 标签。然后，我们让第一行显示当前温度和天气情况：

```
<ion-row>
  <ion-col col-12>
    <h1>{{currentData.temperature | number:'.0-0'}}&deg;</h1>
    <p>{{currentData.summary}}</p>
  </ion-col>
</ion-row>
```

你可以看一下 *data.json* 文件，你会发现气温实际是精确到小数点后两位。我觉得我们的 App 用不到这么高的精度。因此，我们可以使用 Angular 内置的管道操作来进行取整。

**管道是什么？**

管道用以一个数据作为输入，将它转换成符合要求的输出。Angular 包含了几个内置的管道，比如 DatePipe、UpperCasePipe、LowerCasePipe、CurrencyPipe 和 PercentPipe。

通过在数据绑定 `currentData.temperature` 之后添加 **`| number:'.0-0'`**，我们的数据会将 58.59 转成 59。

然后，我们继续添加新行，用来显示未来三天的最低气温和最高气温。

```
<ion-row>
  <ion-col col-4>
    {{daily.data[0].temperatureMax | number:'.0-0'}}&deg;<br>
    {{daily.data[0].temperatureMin | number:'.0-0'}}&deg;
  </ion-col>
  <ion-col col-4>
    {{daily.data[1].temperatureMax | number:'.0-0'}}&deg;<br>
    {{daily.data[1].temperatureMin | number:'.0-0'}}&deg;
  </ion-col>
  <ion-col col-4>
```

```
      {{daily.data[2].temperatureMax | number:'.0-0'}}&deg;<br>
      {{daily.data[2].temperatureMin | number:'.0-0'}}&deg;
    </ion-col>
  </ion-row>
```

保存文件，运行 App，你会遇到一个错误。这是因为我们的数据是以 Promise 的方式加载的，对于模板来说一开始的时候是不可用的。要解决这个问题，我们可以用 `ngIf` 指令告诉模板在加载好数据之前不要显示网格。当然，我们还可以为我们的变量指定初始值，这样模板能够渲染，但非数据绑定的元素则会显示，比如°符号。但如果使用 `ngIf`，我们两者都能控制，因此将 `<ion-grid>` 标签修改成这样：

```
<ion-grid *ngIf="daily.data != undefined">
```

保存文件，我们的模板在数据加载成功后会正确渲染。如图 9-2 所示。

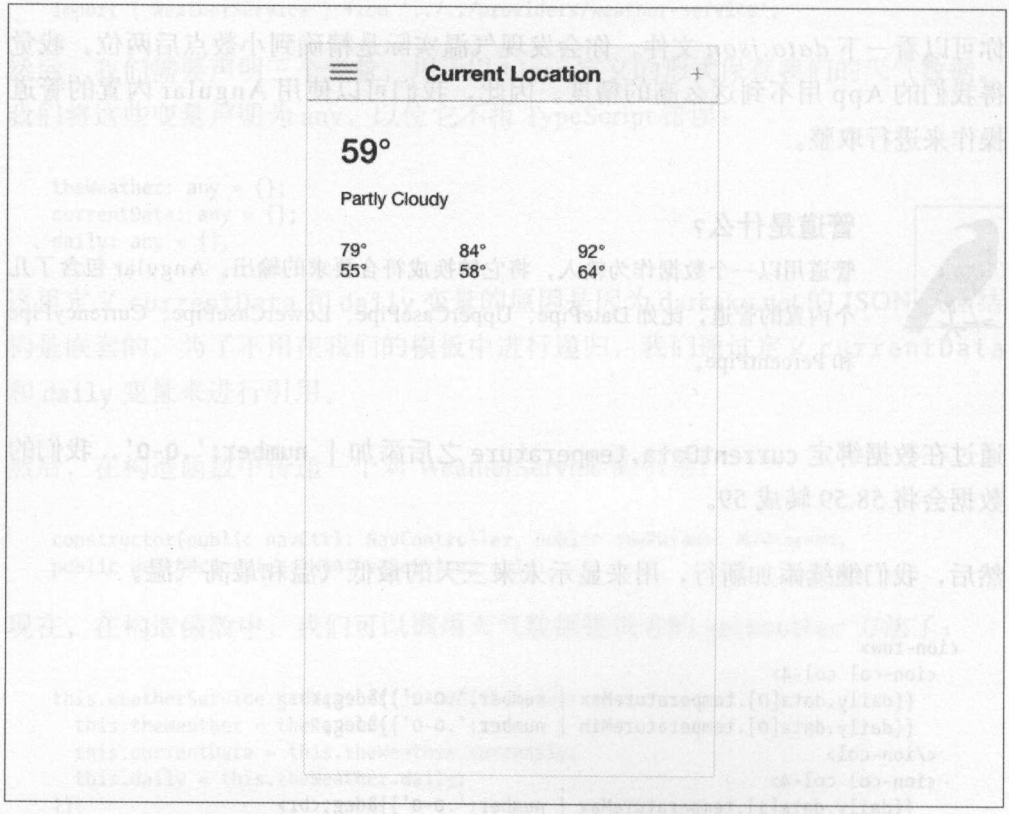



图 9-2：我们模拟的天气数据显示出来了

## 进度显示：loading 对话框和下拉刷新

当 App 加载天气数据时最好能给用户一些提示。尽管在使用模拟数据时加载过程可能很短，但一旦我们将它换成在线数据，它需要的时间可能会变长。

Ionic 有一个 LoadingController 组件，可以很容易添加到我们的 App 中。修改 ionic-angular 的 import 语句，让它包含 Loading。

```
import { NavController, NavParams, LoadingController } from 'ionic-angular';
```

然后，在类定义中，添加一个类型为 Loading 的 loader 变量：

```
export class WeatherPage {
  theWeather: any = {};
  currentData: any = {};
  daily: any = {};
  loader: LoadingController;
```

我们还需要将这个模块也传递到构造函数中去：

```
constructor(public navCtrl: NavController,
            public navParams: NavParams,
            public weatherService: WeatherService,
            public loadingCtrl: LoadingController) {...
```

现在再创建一个 loading 对话框组件实例。在调用 weatherService.getWeather() 方法之前添加代码：

```
let loader = this.loadingCtrl.create({
  content: "Loading weather data...",
  duration: 3000
});
```

为了便于测试，我们将对话框的 duration 设置为三秒，因为目前我们还是用的本地数据。将 duration 设置为三秒，这样在它被隐藏之前我们让它有机会显示出来。要显示 loading 对话框，我们需要告诉 loader 实例用 present 方法显示自己：

```
loader.present();
```

关于 Loading 组件的更多细节，请参考 Ionic 框架官网。

保存文件，运行 App，加载对话框显示三秒后隐藏（见图 9-3）。

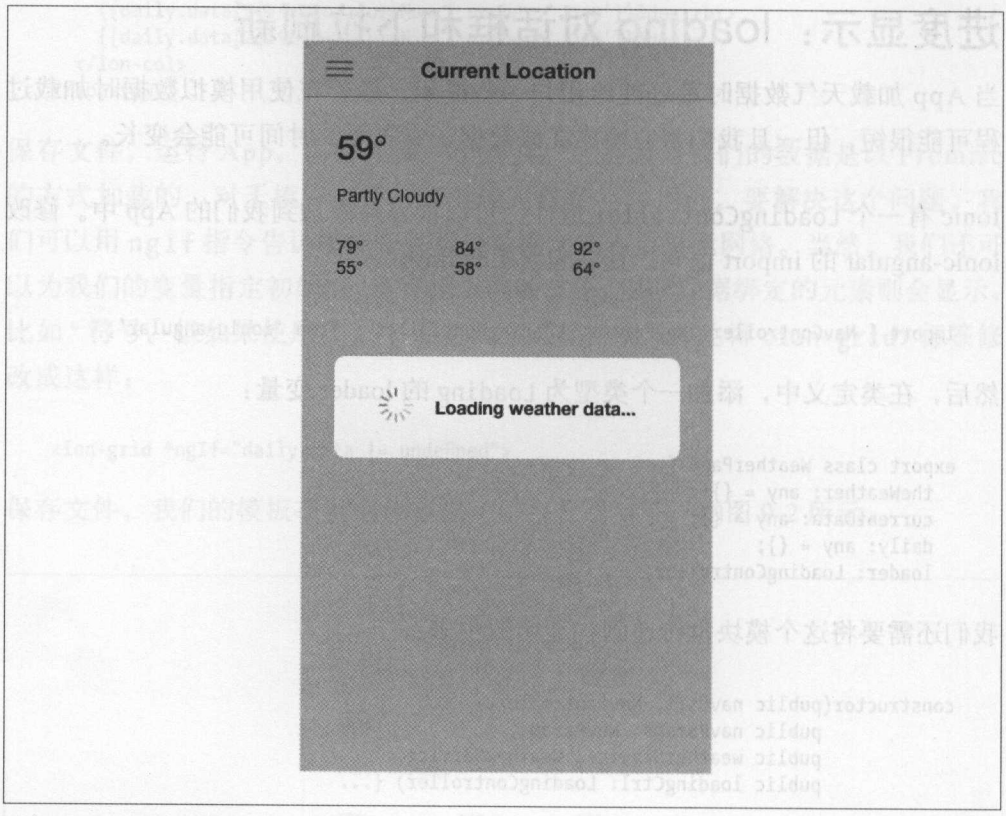



图 9-3：loading 对话框

因为我们在加载数据时使用了对话框，我们也可以使用 Refresher 组件（即下拉刷新）。这是一个常见的 UX 方法，允许用户在不使用任何可视控件的情况下刷新数据。

在 *weather.html* 文件中，在 `<ion-content>` 之后添加：

```
<ion-refresher (ionRefresh)="doRefresh($event)">
  <ion-refresher-content
    pullingIcon="arrow-dropdown"
    pullingText="Pull to refresh"
    refreshingSpinner="circles"
    refreshingText="Refreshing...">
  </ion-refresher-content>
</ion-refresher>
```

这需要定义 `<ion-refresher>` 和 `<ion-refresher-content>`。除了指定要显示

的文字，你还可以指定你想要的旋转动画。内置的动画是 `ios`、`ios-small`、`bubbbles`、`circles`、`crescent` 和 `dots`。

当下拉动作触发 refresh 组件时，会调用 `doRefresh` 方法。我们来为 *weather.ts* 文件添加一个 `doRefresh` 方法。

首先，我们需要在 import 语句中包含这个组件：

```
import { NavController, NavParams, LoadingController, Refresher } from
'ionic-angular';
```

然后在类定义中用一个变量引用 Refresher：

```
export class WeatherPage {
  theWeather: any = {};
  currentData: any = {};
  daily: any = {};
  loader: LoadingController;
  refresher: Refresher;
```

然后在类中添加一个 `doRefresh` 方法：

```
doRefresh(refresher) {
  setTimeout(() => {
    refresher.complete();
  }, 2000);
}
```

这里，我们简单调用了一个 timeout 延迟两秒钟时间。在我们换成在线数据之后，我们还会回到这个方法，目前，只是测试一下 Refresher UI 的效果。

## 添加 Geolocation

Dark Sky API 需要用经纬度作为参数才能获得天气预报。我们可以用 Cordova 的 Geolocation 插件来获取经纬度。要在 App 中添加这个插件，我们需要用这个命令：

```
$ ionic plugin add cordova-plugin-geolocation
```

CLI 会向已安装的平台添加这个插件。注意，这个命令只会安装插件，不会添加要使用这个插件的代码。在 *app.module.ts* 文件中，我们必须 import 它并将 Geolocation 插件提供者添加到 `@NgModule` 声明中。

首先打开 *weather.ts*，从 Ionic Native 中导入 Geolocation：

```
import { Geolocation } from '@ionic-native/geolocation';
```

同时在构造函数中添加对应的参数。

如果你没忘记的话，Ionic Native 的作用相当于是 Cordova 插件的 Angular 封装。

然后，在构造函数中，我们就可以在 Geolocation 模块上调用 getCurrentPosition 函数了。暂时，我们将当前位置的经纬度输出到控制台中就可以了：

```
Geolocation.getCurrentPosition().then(pos => {
  console.log('lat: ' + pos.coords.latitude + ', lon: ' + pos.coords.longitude);
});
```

因为开启GPS比较耗电，我们可以将结果保存起来。我们不将它保存为普通对象，而是定义一个实现了 CurrentLoc 接口的变量。

这样，我们就需要定义一个接口。在 *app* 目录下创建一个 *interfaces* 目录。在这个目录中，新建一个文件，名为 *current-loc.ts*。这个文件用于定义我们的 CurrentLoc 接口，代码如下：

```
export interface CurrentLoc {
  lat: number;
  lon: number;
  timestamp?: number;
}
```

这个接口告诉编译器，这个接口需要两个必须属性：lat 和 lon，二者都是 number 类型。它还有一个可选属性 timestamp。可选属性通过在属性名后添加 ? 来定义。

回到 *weather.ts*，我们必须在我们的类中导入这个接口：

```
import { CurrentLoc } from '../../interfaces/current-loc';
```

现在，我们可以声明一个实现了 CurrentLoc 接口的 currentLoc 变量：

```
currentLoc: CurrentLoc = {lat:0 , lon: 0};
```

我们调用了 getCurrentPosition 方法，然后将结果保存在 currentLoc 变量：

```
Geolocation.getCurrentPosition().then(pos => {
  console.log('lat: ' + pos.coords.latitude + ', lon: ' + pos.coords.longitude);
  this.currentLoc.lat = pos.coords.latitude;
  this.currentLoc.lon = pos.coords.longitude;
  this.currentLoc.timestamp = pos.timestamp;
});
```

我们还保存了 timestamp，这样我们可以检查数据的更新时间并触发一个更新。我将这个任务留给你作为 App 完成后最终的编程练习。现在，我们已经获得了你在地球上的坐标，让我们来查找天气数据。

## 访问在线天气数据

你可以使用不同的天气服务，但对于本 App，我们将使用 Dark Sky 的服务。访问 Dark Sky 注册页面（*https://darksky.net/dev/register*），注册一个账号。

注册完毕，你会获得一个 API key。我们会在 API 中使用这个 API key。目前，这个 API 允许最多每天进行 1000 次调用，超过这个限制就需要进行付费。

这个 API 实际上非常简单：

```
https://api.darksky.net/forecast/APIKEY/LATITUDE,LONGITUDE
```

如果你将我们本地的 *weather-data.ts* 文件替换成这个 URL（用你自己的 API key 替换其中的 APIKEY，以及替换其中的硬编码的坐标），你会发现无法工作。打开 JavaScript 控制台，你会看到类似如下错误：

```
XMLHttpRequest
cannot load https://api.darksky.net/forecast/APIKEY/LATITUDE,LONGITUDE.
No 'Access-Control-Allow-Origin' header is present on the requested resource.
Origin 'http://localhost:8100' is therefore not allowed access.
```

这是由于浏览器安全策略的限制，因为数据默认是不允许进行跨域访问的。也就是所谓的 CORS（Cross Origin Resource Sharing，跨域资源共享）。正常情况下，要在开发期间解决这个问题需要使用代理服务器或者类似的方法。幸好，Ionic CLI 提供了一种现成的解决办法。

在 *ionic.config.json*（在 App 的根目录下）中，我们可以定义一系列代理给 `ionic serve` 使用。只需要添加一个 `proxies` 属性，并指定其 path 属性和 proxyUrl 属性即可。例如，对于目前这个 App，你需要这样做：

```
{
  "name": "Ionic2Weather",
  "app_id": "",
  "v2": true,
  "typescript": true,
  "proxies": [
    {
      "path": "/api/forecast",
      "proxyUrl": "https://api.darksky.net/forecast/APIKEY"
    }
  ]
}
```

将 proxyUrl 中的 APIKEY 替换成你自己的 API key。

保存文件，重新执行 `$ ionic serve`，使修改生效。

返回 *weather-service.ts* 文件，我们可以将 http.get 方法调用修改为：

```
this.http.get('/api/forecast/43.0742365,-89.381011899')...
```

我们的 App 就能够正确调到 Dark Sky API，在线的天气数据也能够返回给我们的 App 了。

## 将 Geolocation 和 Weather 提供者关联

目前，我们的经纬度值在请求 Dark Sky 时是硬编码的。

当然，我们需要获取当前地理位置。我们已经写好了调用 `Geolocation` 函数的代码。因为我们使用了 Ionic Native 包装器来调用 Cordova 插件，这个方法也是以 Promise 的形式调用的。使用 Promise 的好处之一是能够进行链式调用，这里我们就需要用到这样一个链式调用。当我们获取到当前位置时，就可以接着调用 Dark Sky 来获得天气预报。

要在 Promise 上进行链式调用，你可以再调用一个 `.then` 函数：

```
geolocation.getCurrentPosition().then(pos => {
  console.log('lat: ' + pos.coords.latitude + ', lon: ' + pos.coords.longitude);
  this.currentLoc.lat = pos.coords.latitude;
  this.currentLoc.lon = pos.coords.longitude;
  this.currentLoc.timestamp = pos.timestamp;
  return this.currentLoc;
}).then(currentLoc => {
```

```
      weatherService.getWeather(currentLoc).then(theResult => {
      this.theWeather = theResult;
      this.currentData = this.theWeather.currently;
      this.daily = this.theWeather.daily;
      loader.dismiss();
    });
  });
```

除了增加了一个 `.then`，我们还在 Geolocation 的 Promise 中添加了一句 return `this.currentLoc`。通过这种方式，就可以将数据沿 Promise 链传递。

**示例代码警告**

这里我们偷了一小点懒，没有考虑到一个问题。在对任何远程地址进行网络请求的时候，你都应该考虑失败的情况并进行处理。

这个例子极好地说明了通过 Promise 进行异步调用是如何地简单。

另外一个需要修改的小地方，是从我们的 loading 对话框中将 duration 值移除：

```
let loader = this.loadingCtrl.create({
  content: "Loading weather data..."
});
loader.present();
```

我们会在抓取到天气数据时关闭 loading 对话框。这时我们只需要调用 `loading.dismiss()`，就可以隐藏 loading 对话框了。

现在，我们需要修改 *weather-service.ts* 以支持动态的地址。

首先，导入我们自定义的 location 类：

```
import { CurrentLoc } from '../interfaces/current-loc';
```

然后修改 load 函数，以接收当前位置作为参数：

```
load(currentLoc:CurrentLoc) {
```

然后修改 http.get 一句的调用，让它引用这个对象：

```
this.http.get('/api/forecast/'+currentLoc.lat+','+currentLoc.lon)
```

最后，我们还需要修改 getWeather 方法，使它支持一个位置参数：

```
getWeather(currentLoc:CurrentLoc) {
  this.data = null;
  return this.load(currentLoc).then(data => {
    return data;
  });
}
```

我们在加载新的数据之前，清除了上一次从天气服务中请求得来的数据。这是修改后的 *weather-service.ts* 文件：

```
import { Injectable } from '@angular/core';
import { Http } from '@angular/http';
import 'rxjs/add/operator/map';
import { CurrentLoc } from '../interfaces/current-loc'

@Injectable()
export class WeatherService {
  data: any = null;

  constructor(public http: Http) {
    console.log('Hello WeatherService Provider');
  }

  load(currentLoc:CurrentLoc) {
    if (this.data) {
      return Promise.resolve(this.data);
    }

    return new Promise(resolve => {
      this.http.get('/api/forecast/'+currentLoc.lat+','+currentLoc.lon)
        .map(res => res.json())
        .subscribe(data => {
          this.data = data;
          resolve(this.data);
        });
    });
  }

  getWeather(currentLoc:CurrentLoc) {
    this.data = null;
    return this.load(currentLoc).then(data => {
      return data;
    });
  }
}
```

最后还有一个要修改的地方，是将 *app.component.ts* 文件中的 `this.weatherData.load()` 一句删除。保存文件，重新启动 App，在线的天气预报就会显示出来了。

# 获取其他地区的天气

能够查询你所在的地区的天气是很好的，但如果还想知道这个世界上其他地区的天气怎么办？我们将使用侧滑菜单来切换不同的城市，如图 9-4 所示。

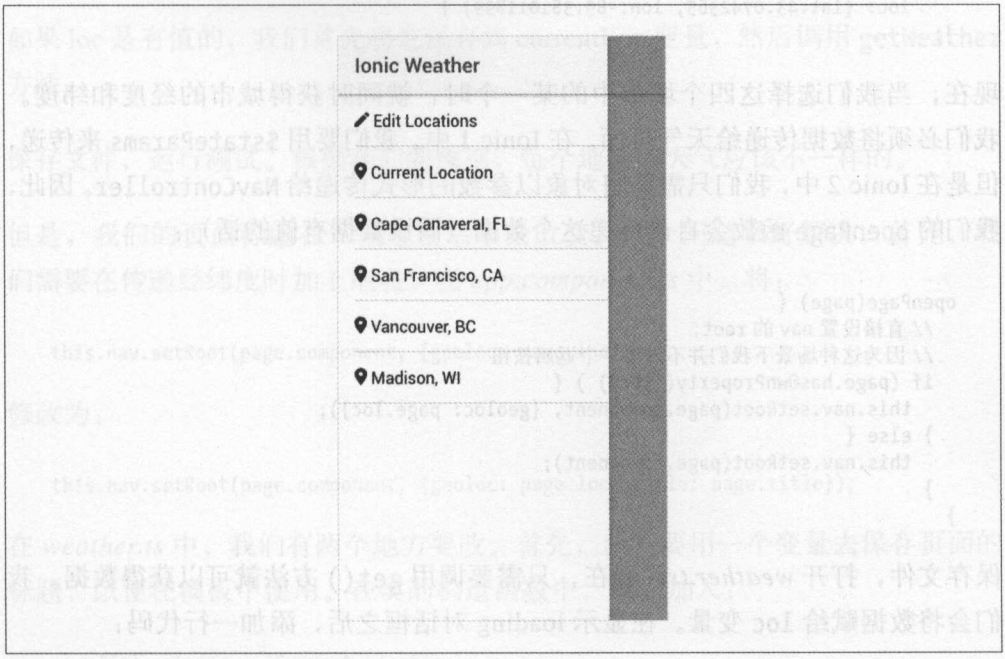



图 9-4：Ionic 天气 App 的侧滑菜单

第一个要修改的地方是在 *app.component.ts* 文件中的 pages 数组。首先，让我们导入 CurrentLoc 类：

```
import { CurrentLoc } from '../interfaces/current-loc';
```

然后，我们需要在数组中加入要获取天气的地区的经纬度。因为对于添加地区和当前地区页面来说，都是无法指定位置的经纬度的，所以我们将新添加的这个属性定义为 optional 可选：

```
pages: Array<{title: string, component: any, icon: string, loc?:CurrentLoc}>;
```

然后，在数组中填入一些地区：

```
this.pages = [
  { title: 'Edit Locations', component: LocationsPage, icon: 'create' },
  { title: 'Current Location', component: WeatherPage, icon: 'pin' },
```

```
    { title: 'Cape Canaveral, FL', component: WeatherPage, icon: 'pin', ↵
      loc: {lat:28.3922, lon:-80.6077} },
    { title: 'San Francisco, CA', component: WeatherPage, icon: 'pin', ↵
      loc: {lat:37.7749, lon:-122.4194} },
    { title: 'Vancouver, BC', component: WeatherPage, icon: 'pin', ↵
      loc: {lat:49.2827, lon:-123.1207} },
    { title: 'Madison, WI', component: WeatherPage, icon: 'pin', ↵
      loc: {lat:43.0742365, lon:-89.381011899} }
];
```

现在，当我们选择这四个城市中的某一个时，就同时获得城市的经度和纬度。我们必须将数据传递给天气页面。在 Ionic 1 中，我们要用 `$stateParams` 来传递，但是在 Ionic 2 中，我们只需要将对象以参数的形式传递给 `NavController`。因此，我们的 `openPage` 函数会自己传递这个数据（假设数据有值的话）：

```
openPage(page) {
  // 直接设置 nav 的 root，
  // 因为这种场景下我们并不需要一个返回按钮
  if (page.hasOwnProperty('loc') ) {
    this.nav.setRoot(page.component, {geoloc: page.loc});
  } else {
    this.nav.setRoot(page.component);
  }
}
```

保存文件，打开 *weather.ts*。现在，只需要调用 `get()` 方法就可以获得数据。我们会将数据赋给 `loc` 变量。在显示 loading 对话框之后，添加一行代码：

```
let loc = this.navParams.get('geoloc');
```

如果这个变量的值为 undefined，我们可以像之前一样调用 `Geolocation`。如果这个值是 defined 的，我们就用这个值来调用天气服务：

```
geolocation.getCurrentPosition().then(pos => {
  console.log('lat: ' + pos.coords.latitude + ', lon: ' + pos.coords.longitude);
  this.currentLoc.lat = pos.coords.latitude;
  this.currentLoc.lon = pos.coords.longitude;
  this.currentLoc.timestamp = pos.timestamp;
  return this.currentLoc;
 }).then(currentLoc => {
   weatherService.getWeather(currentLoc).then(theResult => {
    this.theWeather = theResult;
    this.currentData = this.theWeather.currently;
    this.daily = this.theWeather.daily;
    loader.dismiss();
   });
  });
} else {
  this.currentLoc = loc;
```

```
    weatherService.getWeather(this.currentLoc).then(theResult => {
      this.theWeather = theResult;
      this.currentData = this.theWeather.currently;
      this.daily = this.theWeather.daily;
      loader.dismiss();
    });
  }
```

如果 loc 是有值的，我们首先将它保存到 currentLoc 变量，然后调用 getWeather 方法。

保存文件，运行测试。根据我们的预期，每个地区的天气应该不一样的。

但是，我们的页面标题在切换地区后不会改变。这个问题很好解决。首先，我们需要在传递经纬度时加上地名。在 *app.component.ts* 中，将：

```
this.nav.setRoot(page.component, {geoloc: page.loc});
```

修改为：

```
this.nav.setRoot(page.component, {geoloc: page.loc, title: page.title});
```

在 *weather.ts* 中，我们有两个地方要改。首先，我们要用一个变量去保存页面的标题，以便在模板中使用。在类的构造函数中，我们加入：

```
pageTitle:string = 'Current Location';
```

然后，在对请求其他地方天气预报的代码块中，我们可以从 NavParam 读取数据然后放到 pageTitle 中：

```
this.pageTitle = this.navParams.get('title');
```

最后来修改模板。我们需要在 <ion-title> 标签中使用 pageTitle 变量：

```
<ion-title>{{pageTitle}}</ion-title>
```

这样，我们的页面标题就会显示为当前地区的名字。

# 下拉刷新：第二部分

你可能会奇怪为什么要那么麻烦，将 currentLoc 设置为传进来的地理位置。我们可以直接将它传递到 getWeather 函数中啊。还记得之前添加的 refresh

组件吗？哦，是的。我们到现在还没有让它发挥作用，除了延迟两秒之外。让我们来修改一下代码，真正去拉取新的天气信息。我们所需要做的就是将 setTimeout 替换掉，这样 doRefresh 函数就变成了：

```
doRefresh(refresher) {
  this.weatherService.getWeather(this.currentLoc).then(theResult => {
    this.theWeather = theResult;
    this.currentData = this.theWeather.currently;
    this.daily = this.theWeather.daily;
    refresher.complete();
  });
}
```

### API 限制

在 doRefresh 方法中，最好检查一下 timestamp 属性以防止 API 滥用。

不幸的是，我猜想你可能无法看到天气数据会有任何改变，因为天气总不可能变得有多么快。接下来我们看看编辑地址页面。

## 编辑地址

这个页面允许我们向列表中添加新的城市，或者删除已有的城市。它最终看起来如图 9-5 所示。

打开 *locations.html* 文件，让我们在 <ion-content> 里面添加一个添加城市按钮。同时删除 padding 属性：

```
<ion-content>
  <button ion-button icon-left clear color="dark" item-left
  (click)="addLocation()">↵
    <ion-icon name="add"></ion-icon>Add City</button>
</ion-content>
```

对于这个按钮，我们在它上面使用 clear 样式和 dark 样式。clear 属性将删除按钮的边框和背景，而 dark 属性会为按钮添加一点加暗效果。我们还让 click 事件和 addLocation 函数绑定。

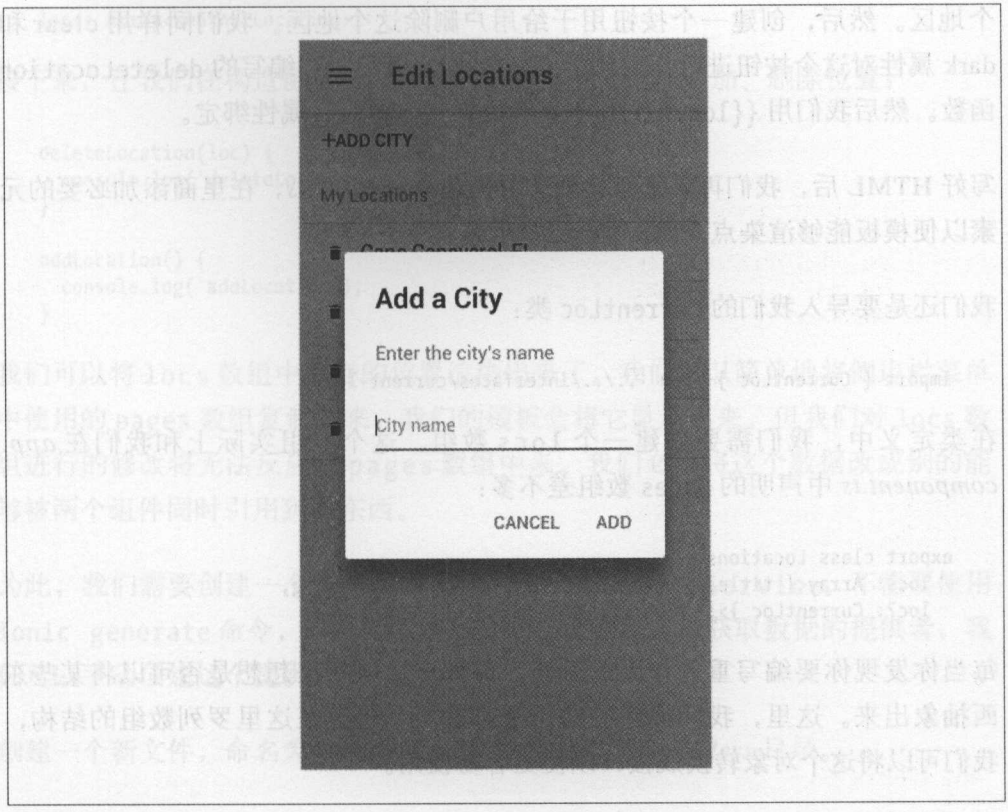



图 9-5：添加城市对话框

在 <button> 标签后面，我们添加了 <ion-list> 标签用于显示已保存的位置列表：

```
<ion-list>

  <ion-list-header>
    My Locations
  </ion-list-header>

  <ion-item *ngFor="let loc of locs">
   <button ion-button icon-left clear color="dark"
   (click)="deleteLocation(loc)">
     <ion-icon name="trash"></ion-icon>
    </button>{{loc.title}}
  </ion-item>

</ion-list>
```

我们用 <ion-list-header> 来显示列表头而不是用常规的 header 标签。幸好，<ion-item> 标签对你来说已经很熟悉了。它会遍历一个 locs 数组并挨个显示每

个地区。然后，创建一个按钮用于给用户删除这个地区。我们同样用 clear 和 dark 属性对这个按钮进行样式化。click 事件会调用我们编写的 deleteLocation 函数。然后我们用 {{loc.title}} 来将位置名称和 text 属性绑定。

写好 HTML 后，我们再来修改代码。首先是 *locations.ts*，在里面添加必要的元素以便模板能够渲染点东西。

我们还是要导入我们的 CurrentLoc 类：

```
import { CurrentLoc } from '../../interfaces/current-loc';
```

在类定义中，我们需要创建一个 locs 数组。这个数组实际上和我们在 *app.component.ts* 中声明的 pages 数组差不多：

```
export class Locations {
  locs: Array<{ title: string, component: any, icon: string,
  loc?: CurrentLoc }>;
```

每当你发现你要编写重复代码的时候，你都应该停下来想想是否可以将某些东西抽象出来。这里，我们得到的结论是“是”。代替在这里罗列数组的结构，我们可以将这个对象转换成接口以便更容易使用。

创建新文件，命名为 *weather-location.ts*，放到 *interfaces* 目录。这个目录是我们定义 WeatherLocation 接口的地方。因为 loc 属性使用的是 CurrentLoc 接口，我们也要导入这个模块：

```
import { CurrentLoc } from './current-loc';

export interface WeatherLocation {
  title: string;
  component: any;
  icon: string;
  loc?: CurrentLoc;
}
```

这个接口定义了位置的名称、它的页面组件、图标及一个可选的 CurrentLoc 属性。

创建好接口之后，回到 *locations.ts*，导入这个接口：

```
import { WeatherLocation } from '../../interfaces/weather-location';
```

然后，我们就可以修改 locs 数组的类型了：

```
locs: Array<WeatherLocation>;
```

接下来，让我们在构造函数后面添加两个空函数，用于添加、删除位置：

```
deleteLocation(loc) {
  console.log('deleteLocation');
}

addLocation() {
  console.log('addLocation');
}
```

我们可以将 `locs` 数组中保存的位置渲染出来了。我们可以简单地将侧边栏菜单中使用的 `pages` 数组复制过来，我们的模板会将它显示出来。但我们对 `locs` 数组进行的修改将无法反应到 `pages` 数组中来。我们必须将这个数据改成别的能够被两个组件同时引用到的东西。

为此，我们需要创建一个新的提供者，名为 `LocationsService`。不需要使用 `ionic generate` 命令，这个命令是用来创建需要从远程获取数据的提供者，我们可以手动创建这个提供者。

创建一个新文件，命名为 *locations-service.ts*，放到 *providers* 目录。

首先需要定义 import 语句。我们要用到 Angular 库的 `Injectable`，以及我们自己的 `WeatherLocation` 和 `WeatherPage` 组件：

```
import { Injectable } from '@angular/core';
import { WeatherLocation } from '../interfaces/weather-location';
import { WeatherPage } from '../pages/weather/weather';
```

然后，我们要使用 `@Injectable()` 修饰符：

```
@Injectable()
```

现在来定义我们的服务，`locations` 数组（类型定义为 `WeatherLocation`），然后用默认的地址初始化这个数组：

```
export class LocationsService {
  locations: Array<WeatherLocation>;

  constructor() {
    this.locations = [
      { title: 'Cape Canaveral, FL', component: WeatherPage, icon: 'pin', ↵
        loc: { lat: 28.3922, lon: -80.6077 } },
```

```
      { title: 'San Francisco, CA', component: WeatherPage, icon: 'pin', ↵
        loc: { lat: 37.7749, lon: -122.4194 } },
      { title: 'Vancouver, BC', component: WeatherPage, icon: 'pin', ↵
        loc: { lat: 49.2827, lon: -123.1207 } },
      { title: 'Madison, WI', component: WeatherPage, icon: 'pin', ↵
        loc: { lat: 43.0742365, lon: -89.381011899 } }
  ];
}
```

我们可以在我们的服务中封装三个函数：一个函数用于获取位置，一个函数用于添加位置，一个函数用于删除位置：

```
getLocations() {
    return Promise.resolve(this.locations);
}

removeLocation(loc:WeatherLocation) {
    let index = this.locations.indexOf(loc)
    if (index != -1) {
        this.locations.splice(index, 1);
    }
}

addLocation(loc: WeatherLocation) {
    this.locations.push(loc);
}
```

准备好提供者后，我们必须将它们导入 *app.module.ts* 的 provider 数组中：

```
import { LocationsService } from '../providers/locations-service';
...
providers: [
  StatusBar,
  SplashScreen,
  Geolocation,
  { provide: ErrorHandler, useClass: IonicErrorHandler },
  WeatherService,
  GeocodeService,
  LocationsService
]
```

返回 *locations.ts*，我们可以将这个服务导入到组件中了：

```
import { LocationsService } from '../../providers/locations-service';
```

我们还需要将这个模块添加到构造函数中，调用这个服务的 `getLocations` 方法读取默认的位置，最终保存到我们的 `locs` 数组：

```
constructor(public navCtrl: NavController,
            public navParams: NavParams,
```

```
            public locationsService: LocationsService) {
    locationsService.getLocations().then(res => {
      this.locs = res;
    });
  }
```

## 删除城市

让我们来真正实现 *locations.ts* 文件中的 `deleteLocation` 函数。在我们的组件中已经有一个这个函数的空实现。我们需要做的就是在这个方法中调用 `LocationsService` 的 `removeLocation` 方法：

```
deleteLocation(loc:WeatherLocation) {
  this.locationsService.removeLocation(loc);
}
```

## 添加城市

添加城市要稍微复杂一些。你应该记得，Dark Sky 的天气服务使用经纬度来查找天气预报。你通常不会这样问：“经纬度为 32.715,-117.1625 的天气如何？”而是会问：“圣迭戈的天气如何？”要在地名和对应的经纬度之间进行转换，我们必须使用 geocoding（地理编码）服务。

# 使用 Geocoding 服务

对于本 App 而言，我们将使用 Google 地图的 Geocoding API。

要使用这个 API，我们必须注册一个 Google 开发者账号并为我们的 App 申请 API key。打开 Google 地图 API 页，根据页面说明生成一个可以使用 geocoding API 的 API key。记下这个 40 位字符的字符串，等下我们会用到它：

```
export class GeocodeService {

  data: any;

  apikey:String = 'YOUR-API-KEY-HERE';
  constructor(public http: Http) {
    this.data = null;
  }

  getLatLong(address:string) {
    if (this.data) {
```

```
      // 已经加载数据
      return Promise.resolve(this.data);
    }

    // 未加载数据
    return new Promise(resolve => {
      this.http.get('https://maps.googleapis.com/maps/api/geocode/↵
      json?address='+encodeURIComponent(address)+'&key='+this.apikey)
        .map(res => res.json())
        .subscribe(data => {
          if(data.status === "OK") {
            resolve({name: data.results[0].formatted_address, location:{
                      latitude: data.results[0].geometry.location.lat,
                      longitude: data.results[0].geometry.location.lng
            }});
          } else {
            console.log(data);
            // 拒绝
          }
        });
    });
  }
}
```

第一个改变的地方是增加了一个新的变量 `apikey`。你需要将它的值设置为你自己的 API key。

在构造函数中，将 data 变量设置为 `null`。这个类最核心的部分实际上是 `getLatLong` 方法。这个方法接受一个 `address` 参数。

这个方法会发送一个请求：

```
this.http.get('https://maps.googleapis.com/maps/api/geocode/json?↵
address='+encodeURIComponent(address)+'&key='+this.apikey)
```

注意，在调用 Google 地图 Geocoding API 之前，我们必须用 `encodeURIComponent` 方法对地址字符串进行编码。

当数据从服务器返回，我们应该用 `data.status` 判断数据中是否找到了一个对应的地址。然后我们就可以检索 JSON，读取所需的数据，然后解决掉这个承诺。

现在我们已经能够将圣迭戈转换成 32.715,-117.1625 了，我们可以回到 *locations.ts* 并完成我们的 `addLocation` 函数了。

我们需要导入我们的 geocode 服务：

```
import { GeocodeService } from '../../providers/geocode-service';
```

然后在构造函数中加入它：

```
constructor(public navCtrl: NavController,
            public navParams: NavParams,
            public locationsService: LocationsService,
            public geocodeService: GeocodeService) {
```

这次我们不用 Ionic Native 对话框插件来弹出用户输入兴趣点的对话框，我们用 AlertController 组件。我们同样需要导入这个库：

```
import { NavController, NavParams, AlertController } from 'ionic-angular';
```

使用 Ionic Native Alert 的难点是你可以扩展它。这个 Alert 必须在模拟器或设备上才能测试。而标准的 Ionic Alert 组件则可以直接在浏览器中进行开发。但是，和 Ionic Native 对话框不同，它需要我们书写更多的 JavaScript 代码。我们需要修改构造函数，让它包含 AlertController：

```
constructor(public navCtrl: NavController,
            public navParams: NavParams,
            public locationsService: LocationsService,
            public geocodeService: GeocodeService,
            public alertCtrl: AlertController) {
```

这是使用了 Alert 组件之后的 addLocation 方法：

```
addLocation() {
  let prompt = this.alertCtrl.create({
    title: 'Add a City',
    message: "Enter the city's name",
    inputs: [
      {
        name: 'title',
        placeholder: 'City name'
      },
    ],
    buttons: [
      {
        text: 'Cancel',
        handler: data => {
        console.log('Cancel clicked');
      }
    },
    {
      text: 'Add',
      handler: data => {
        console.log('Saved clicked');
```

```
        }
      }
    ]
    });

    prompt.present();
  }
```

关键的代码是 prompt.present()，告诉 AlertController 显示我们的 alert。

现在，我们还不会对你输入的城市做任何事情，需要继续添加代码。将这一句：

```
console.log('Saved clicked');
```

替换为：

```
if (data.title != '') {
  this.geocodeService.getLatLong(data.title).then(res => {
    let newLoc = { title: '', component: WeatherPage, icon: 'pin',
                   loc: { lat: 0, lon: 0 } }
    newLoc.title = res.name;
    newLoc.loc.lat = res.location.latitude;
    newLoc.loc.lon = res.location.longitude;

    this.locationsService.addLocation(newLoc);
  });
}
```

我们还需要导入 Weather 组件：

```
import { WeatherPage } from '../weather/weather';
```

现在，我们可以将城市名称转换成经纬度，然后添加到我们的位置列表里。

在这段代码中，有几个地方值得探讨。首先是我们没有处理 geodcoding 失败的情况。如果你打算挑战一下自己，你可以添加拒绝承诺的代码。第二个问题是 geocoding 返回的结果不止一个的情况。例如，我输入了 Paris，那么我的意思是 Paris、France 或 Paris、Texas？对于这种情况，你可能需要用一个更复杂的 Alert 组件，上面显示一个单选按钮列表。

## 动态刷新侧滑菜单

如果你添加了一个新地址到地区列表，然后试图用侧滑菜单查看某个地区的天气，你会发现列表中那个地区根本没列出来。这是因为列表引用的还是本地版本，

而不是 LocationsService 中的数组。让我们打开 *app.component.ts* 文件，解决这个问题。

首先，我们需要导入这个服务：

```
import { LocationsService } from '../providers/locations-service';
```

接着，我们将它添加到构造函数：

```
constructor(
  public platform: Platform, public locationsService: LocationsService
) {
  this.initializeApp();
  this.getMyLocations();
}
```

getMyLocations 函数应该从 LocationsService 提供者获得数据并填充 pages 数组：

```
getMyLocations(){
  this.locationsService.getLocations().then(res => {
    this.pages = [
      { title: 'Edit Locations', component: LocationsPage, icon: 'create' },
      { title: 'Current Location', component: WeatherPage, icon: 'pin' }
    ];
    for (let newLoc of res) {
      this.pages.push(newLoc);
    }
  });
}
```

保存文件，重新运行 $ ionic serve。一开始侧滑菜单中显示的仍然是最初的位置列表。当我们添加一个新位置，Edit Location 界面会正确地更新。但是，侧滑菜单仍然不显示我们添加的新地址。尽管侧滑菜单也是从我们共用的提供者获取的数据，但它完全不知道这些数据什么时候发生了改变。要解决这个问题，我们需要学习两个截然不同的内容：Ionic 事件和 Observable。你可能需要把当前的工作进度放一放，因为这两个问题哪个都不简单。

## Ionic 事件

第一种办法是使用 Ionic 事件以获得数据改变通知。这里引用一段官方文档：

事件是一种发布－订阅事件的机制，负责处理App中的应用程序级别的事件。

听起来和我们想要达到的目的非常吻合。在*locations.ts*文件中，我们需要从ionic-angular中导入Events组件：

```
import {NavController, NavParams, AlertController, Events} from 'ionic-angular';
```

在构造函数中，我们将Events模块加入：

```
constructor(public navCtrl: NavController,
            public navParams: NavParams,
            public locationsService: LocationsService,
            public geocodeService: GeocodeService,
            public alertCtrl: AlertController,
            public events: Events) {
```

现在，在deleteLocation和addLocation函数中，我们需要添加事件发布的代码：

```
this.events.publish('locations:updated', {});
```

这样，deleteLocation函数会变成：

```
deleteLocation(loc) {
  this.locations.removeLocation(loc);
  this.events.publish('locations:updated', {});
}
```

在addLocation函数中，在Add回调块中：

```
handler: data => {
  if (data.title != '') {
    this.geocode.getLatLong(data.title).then(res => {
      let newLoc = { title: '', component: WeatherPage, icon: 'pin',
                     loc: { lat: 0, lon: 0 } }
      newLoc.title = res.name;
      newLoc.loc.lat = res.location.latitude;
      newLoc.loc.lon = res.location.longitude;

      this.locations.addLocation(newLoc);
      this.events.publish('locations:updated', {});
    });
  }
}
```

publish函数参数包括事件名（即locations:updated）和一个data对象，你需

要传递的数据。就我们的 App 来说，我们不需要向订阅者传递任何数据。

订阅者应当在 *app.component.ts* 文件中添加。首先，我们需要修改 import 语句为：

```
import { Nav, Platform, Events } from 'ionic-angular';
```

然后将 Events 通过构造函数传递：

```
constructor(public platform: Platform,
            public locationsService: LocationsService,
            public events: Events) {
```

在构造函数中，我们要添加这些代码：

```
this.initializeApp();
this.getMyLocations();
events.subscribe('locations:updated', (data) => {
  this.getMyLocations();
});
```

这里监听了 location:updated 事件。然后在事件处理块中调用了 getMyLoations 函数。这样，我们的数组就会被更新，侧滑菜单能够展示最新数据。

对于这个问题，Ionic Events 可能还不是最优方案，但通过它能够让我们了解事件是如何在 App 中进行传播的。

## Observable

你可能好奇，有没有一种方法允许我们不手动发送事件就可以让数据更新事件在 App 中传播。事实上，我们确实有这种方法可用。在 RxJS 库中有一个成员就是 Observable。这是来自官方的一段文档：

> Observer 和 Observable 接口提供了一种基于推送通知的一般机制，即所谓的观察者设计模式。Observable 对象代表了通知的发送者（提供者）；Observer 对象代表了接收通知的类（观察者）。

换句话说，我们用 Ionic 事件来编写的事件通知逻辑完全是可被替换的。以我们的例子为例，有许多代码我们都不需要写了。想象一下许多复杂得多的 App，消息架构是个很棘手的问题。

现在，对于我们来说 RxJS 是一个强大而复杂的库。但我们需要在这个 App 中要做的事情却很简单。首先，我们要创建一个 RxJS Subject，即 Behavior Subject。它用于容纳我们要监听变化的数据。然后，我们需要创建真正的 Observable，用于对 subject 的改变进行监听。如果它观察到改变了，它会广播新的数据。第三也就是最后一个是订阅 `Observable`。当它们绑定在一起，我们的数据就会一直保持最新的版本。

我们开始修改 *locations-service.ts* 文件。将我们的 App 从 Ionic Events 版本替换为 `Observable` 版本，这是我们需要改动的主要文件。

再一次，我们必须导入相应的模块：

```
import { Observable, BehaviorSubject } from 'rxjs/Rx';
```

在 RxJS 库中，有几个不同的 Subject 类型。`BehaviorSubject` 最满足我们的需求，因为它一发现数据就会发送一个 update 消息。其他类型的 Subject 还需要调用另一个方法才能广播 update。

然后，我们需要在这个类中定义真正的 `BehaviorSubject` 和 `Observable`：

```
locations: Array<WeatherLocation>;
locationsSubject: BehaviorSubject<Array<WeatherLocation>> =
new BehaviorSubject([]);
locations$: Observable<Array<WeatherLocation>> =
this.locationsSubject.asObservable();
```

### 变量的命名

对 `Observable` 数据类型的变量进行命名时，根据命名规范，变量名后面需要以 $ 结束。要了解更多 Angular 最佳实践，请阅读 John Papa 的 Style Guide（*http://bit.ly/2lva17N*）。

我们只需要将 `locations$` 设置为 `this.locationsSubject.asObservable()`，就可以将 `locations$` 和 `locationsSubject` 关联起来。locationsSubject 完全不知道 locations 数组的数据。要解决这个问题，可以将 locations 数组作为参数传递给 locationsSubject 的另外一个方法：

```
this.locationsSubject.next(this.locations);
```

这样，我们的 Observable 会发出一个 update 事件，所有引用都会被改变。为了不想在 locations 数组一被改变就重写这句代码，我们可以将它封装成一个 refresh 函数：

```
refresh() {
  this.locationsSubject.next(this.locations);
}
```

在构造函数中，addLocation 函数和 removeLocation 方法中，我们在一改变 locations 数组之后都要调用这个方法。这是完整代码：

```
import { Injectable } from '@angular/core';
import { WeatherLocation } from '../interfaces/weather-location';
import { Weather } from '../pages/weather/weather';
import { Observable, BehaviorSubject } from 'rxjs/Rx';

@Injectable()

export class LocationsService {
  locations: Array<WeatherLocation>;
  locationsSubject: BehaviorSubject<Array<WeatherLocation>> =
  new BehaviorSubject([]);
  locations$: Observable<Array<WeatherLocation>> =
  this.locationsSubject.asObservable();

  constructor() {
    this.locations = [
      { title: 'Cape Canaveral, FL', component: Weather, icon: 'pin', ↵
        loc: { lat: 28.3922, lon: -80.6077 } },
      { title: 'San Francisco, CA', component: Weather, icon: 'pin', ↵
        loc: { lat: 37.7749, lon: -122.4194 } },
      { title: 'Vancouver, BC', component: Weather, icon: 'pin', ↵
        loc: { lat: 49.2827, lon: -123.1207 } },
          { title: 'Madison, WI', component: Weather, icon: 'pin', ↵
            loc: { lat: 43.0742365, lon: -89.381011899 } }
      ];
      this.refresh();
    }

    getLocations() {
      return Promise.resolve(this.locations);
    }

    removeLocation(loc) {
      let index = this.locations.indexOf(loc)
      if (index != -1) {
        this.locations.splice(index, 1);
        this.refresh();
      }
    }
```

```
    addLocation(loc) {
      this.locations.push(loc);
      this.refresh();
    }

    refresh() {
      this.locationsSubject.next(this.locations);
    }
}
```

将我们的服务转变成 `Observable` 之后，我们就该来创建数据订阅者了。在 *locations.ts* 中，我们将这句：

```
locationsService.getLocations().then(res => {
  this.locs = res;
});
```

替换为：

```
locationsService.locations$.subscribe( ( locs: Array<WeatherLocation> ) => {
  this.locs = locs;
});
```

现在我们的 `locs` 数组在 service 的数据发生改变时自动更新。找到并删除两个 `this.event.publish` 语句。

*app.component.ts* 文件同样需要修改。首先，需要删除事件监听器：

```
events.subscribe('locations:updated', (data) => {
    this.getMyLocations();
});
```

然后，我们要在 import 语句中加入 `WeatherLocation` 接口：

```
import { WeatherLocation } from '../interfaces/weather-location';
```

最后，我们将 `getMyLocations` 函数从：

```
getMyLocations(){
  this.locationsService.getLocations().then(res => {
    this.pages = [
      { title: 'Edit Locations', component: LocationsPage, icon: 'create' },
      { title: 'Current Location', component: WeatherPage, icon: 'pin' }
    ];
    for (let newLoc of res) {
      this.pages.push(newLoc);
    }
```

```
    });
  }
```

修改为：

```
getMyLocations(){
  this.locationsService.locations$.subscribe( ( locs: Array<WeatherLocation> ) =>
  {
        this.pages = [
          { title: 'Edit Locations', component: LocationsPage, icon: 'create' },
          { title: 'Current Location', component: WeatherPage, icon: 'pin' }
        ];
        for (let newLoc of locs) {
          this.pages.push(newLoc);
        }
    } );
}
```

这样，我们的天气 App 就迁移到了使用 RxJS Observable 的版本了。Observable 在使用动态数据时是一种很好的解决方案。你应该花点时间学习一下它的使用。接下来将对我们的 App 进行一些美化。

## 调整 App 的样式

我们的 App 现在已经能够运行良好了，我们可以花点时间来调整下样式。在源文件中包含有一张漂亮的有云彩的天空图片。我们将使用这张图片作为我们的背景图。因为我们的 Ionic App 是基于 HTML 和 CSS 的，可以利用原有的 CSS 技能来样式化我们的 App。唯一的挑战是，在 Ionic 中不包含需要应用我们的 CSS 的 HTML 结构。

使用 Ionic 中经过改进的选择器系统，要找到特定的 HTML 元素其实很简单。我们准备将一张漂亮的天空图片作为天气页面的背景图。打开 *weather.scss*，添加下列 CSS：

```
page-weather {
  ion-content{
    background: url(../assets/imgs/bg.jpg) no-repeat center center fixed;
    -webkit-background-size: cover;
    background-size: cover;
  }
}
```

### 编写 Sass

上面添加的 CSS 应当嵌套在 page-weather{} 中间。

现在，让我们将基准字体加大，以便文字更容易阅读：

```
ion-content{
  background: url(../assets/imgs/bg.jpg) no-repeat center center fixed;
  -webkit-background-size: cover;
  background-size: cover;
  font-size: 24px;
}
```

当然，默认的黑色字体和蓝天白云真的不太搭。我们调整 `ion-col` 和它的两个子元素。我们将颜色修改为白色，文字居中，并使用一个带一小点透明的文字阴影：

```
ion-col, ion-col h1, ion-col p {
    color: #fff;
    text-align: center;
    text-shadow: 3px 3px 3px rgba(0,0, 0, 0.4);
}
```

接着，修改当前天气信息的文本：

```
h1 {
    font-size: 72px;
}

p {

    font-size: 36px;
    margin-top: 0;
}
```

当然，还有 header 呢？它看起来有点土而且别扭。我们来添加一个新的类 opaque，并在 *weather.html* 文件的 <ion-header> 标签上使用它。

在 *weather.scss* 文件中，我们将应该用几个类来使我们的 header 显得更洋气。第一个类，我们会使用新的 backdrop-filter 方法：

```
.opaque {
  -webkit-backdrop-filter: saturate(180%) blur(20px);
  backdrop-filter: saturate(180%) blur(20px);
}
```

不幸的是，backdrop-filter 目前只支持 Safari。这样我们就不得不再创建一个向下兼容的方案：

```
.opaque .toolbar-background {
  background-color: rgba(#f8f8f8, 0.55);
}
```

有了这两个 CSS 类之后，我们的设计风格就更显得时髦了。

但是在这个页面上还有其他元素需要样式化，比如 `Refresher` 组件。这和之前的组件相比稍微复杂一点。因为，它有好几个状态，每个状态需要进行样式化。

首先是两个文本元素：pulling 文字和 `refreshing` 文字。不幸的是，当前文档没有指出它们的结构和样式化方法。但是通过在 Chrome 的开发工具中进行检查，这两个元素都应用了 CSS 样式。因此，我们可以添加下面的代码：

```
.refresher-pulling-text, .refresher-refreshing-text  {
    color:#fff;
}
```

小箭头图标也很容易搞定：

```
.refresher-pulling-icon {
    color:#fff;
}
```

但 spinner 怎么弄？暗色的圆圈在我们的背景下实在是不显眼。这个元素的样式化也要花点心思。spinner 实际上是一个 SVG 元素。也就是说不能修改 `<ion-spinner>` 的 CSS，而是要修改 SVG 中的值。需要注意的一点是，SVG 元素中某些属性的名字和 CSS 中的是不一样的。例如，SVG 使用 stroke 来代替 *border*，用 *fill* 来代替 *background-color*。

如果你的 `refreshingSpinner` 属性用的是 circles，那么它的样式应该是：

```
.refresher-refreshing .spinner-circles circle{
    fill:#fff;
}
```

但是，如果你想用半月图形作为你的 spinner，那么它的样式应该是：

```
.refresher-refreshing .spinner-crescent circle{
    stroke:#fff;
}
```

还有最后一个地方需要修改。当 Refresher 显示的时候，你可能没有看见，在 Refresher 和内容区之间有一条 1 像素的白线。在使用一张全屏背景图的时候，这种设计不太合适。我们再一次在 Chrome 开发工具中进行检查，找到要设置的 CSS 属性。这样，我们可以将它修改为：

```
.has-refresher > .scroll-content {
    border-width: 0;
}
```

这是完整的 *weather.scss* 代码：

```
page-weather {
  ion-content{
    background: url(../assets/imgs/bg.jpg) no-repeat center center fixed;
    -webkit-background-size: cover;
    background-size: cover;
    font-size: 24px;
  }

  ion-col, ion-col h1, ion-col p {
    color: #fff;
    text-align: center;
    text-shadow: 3px 3px 3px rgba(0,0, 0, 0.4);
  }

  h1 {
    font-size: 72px;
  }

  p{
    font-size: 36px;
    margin-top: 0;
  }

  .opaque {
    -webkit-backdrop-filter: saturate(180%) blur(20px);
    backdrop-filter: saturate(180%) blur(20px);
  }

    .opaque .toolbar-background {
      background-color: rgba(#f8f8f8, 0.55);
  }

    .refresher-pulling-text, .refresher-refreshing-text  {
      color:#fff;
  }

    .refresher-pulling-icon {
      color:#fff;
  }
```

```
    .refresher-refreshing .spinner-circles circle{
      fill:#fff;
    }

    .has-refresher > .scroll-content {
      border-top-width: 0;
    }
  }
```

经过样式化后的天气 App 如图 9-6 所示。

我们的天气页面看起来很不错。但我们还可以让这个设计更出彩。来点漂亮的图标怎么样？

## 添加天气图标

Dark Sky 返回的数据中其实包含有一个 icon 值，而 Ionicon 也支持了几种天气图标。表 9-1 列出了每个图标的映射关系。

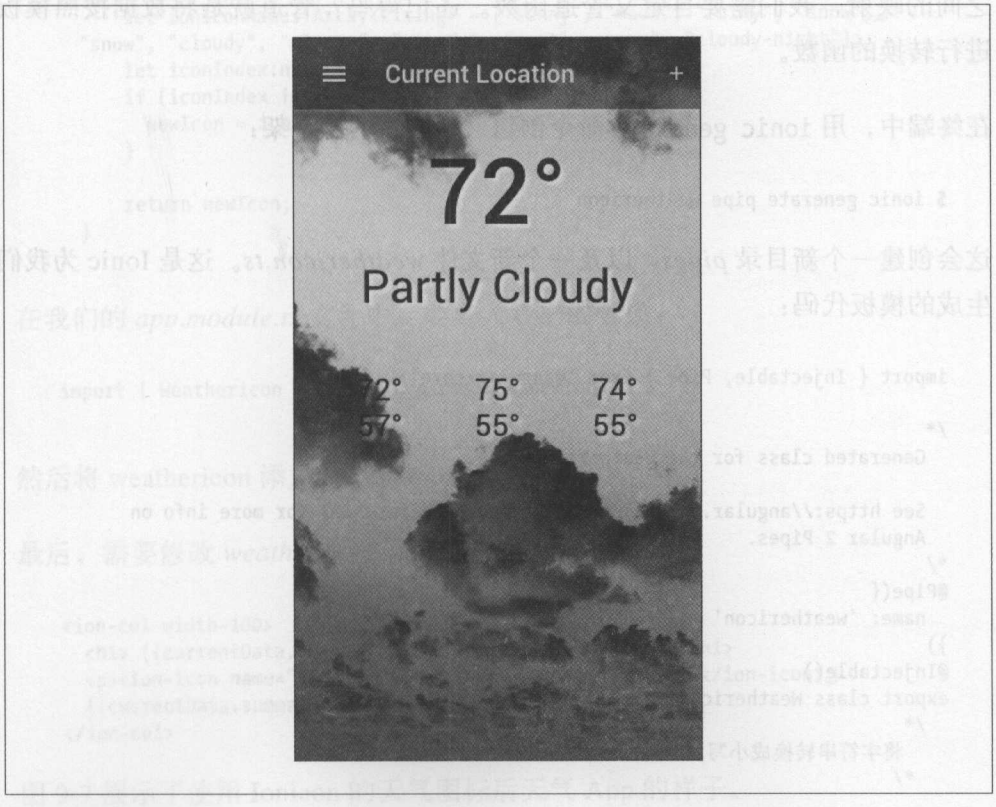



图 9-6：经过样式化后的 Ionic 天气 App

表 9-1：Dark Sky 数据中的图标映射

| Dark Sky 图标名 | Ionicon 图标名 |
| --- | --- |
| clear-day | sunny |
| clear-night | moon |
| rain | rainy |
| snow | snow |
| sleet | snow |
| wind | cloudy |
| fog | cloudy |
| cloudy | cloudy |
| partly-cloudy-day | partly-sunny |
| partly-cloudy-night | cloudy-night |

我们没有 1:1 映射，但这样就够了。要进行 Dark Sky 图标名和 Ionicon 图标名之间的映射，我们需要自定义管道函数。还记得吗？管道就是将数据按照模板进行转换的函数。

在终端中，用 `ionic generate` 命令创建这个管道的脚手架：

```
$ ionic generate pipe weathericon
```

这会创建一个新目录 *pipes*，以及一个新文件 *weathericon.ts*。这是 Ionic 为我们生成的模板代码：

```
import { Injectable, Pipe } from '@angular/core';

/*
  Generated class for the Weathericon pipe.

  See https://angular.io/docs/ts/latest/guide/pipes.html for more info on
  Angular 2 Pipes.
*/
@Pipe({
  name: 'weathericon'
})
@Injectable()
export class Weathericon {
  /*
    将字符串转换成小写。
   */
```

```
  transform(value, args) {
    value = value + ''; // make sure it's a string
    return value.toLowerCase();
  }
}
```

我们需要修改 `transform` 函数中的代码。这个函数的作用是将一个 Dark Sky 图标名转换成对应的 Ionicon 图标名。这是修改后的管道：

```
import { Injectable, Pipe } from '@angular/core';

@Pipe({
  name: 'weathericon'
})
@Injectable()
export class Weathericon {

    transform(value: string, args: any[]) {
      let newIcon:string ='sunny';
      let forecastNames:Array<string> = ["clear-day", "clear-night", "rain", ↵
  "snow", "sleet", "wind", "fog", "cloudy", "partly-cloudy-day",↵
  "partly-cloudy-night"];
      let ioniconNames:Array<string> = ["sunny", "moon", "rainy", "snow",↵
  "snow", "cloudy", "cloudy", "cloudy", "partly-sunny", "cloudy-night"];
      let iconIndex:number = forecastNames.indexOf(value);
      if (iconIndex !== -1) {
        newIcon = ioniconNames[iconIndex];
      }

      return newIcon;
  }
}
```

在我们的 *app.module.ts* 文件中，要导入我们的管道：

```
import { Weathericon } from '../pipes/weatherIcon';
```

然后将 weathericon 添加到 declarations 数组中。

最后，需要修改 *weather.html* 文件的标签：

```
<ion-col width-100>
  <h1> {{currentData.temperature | number:'.0-0'}}&deg;</h1>
  <p><ion-icon name="{{currentData.icon | weathericon}}"></ion-icon>↵
  {{currentData.summary}}</p>
</ion-col>
```

图 9-7 展示了使用 Ionicon 的天气图标后天气 App 的样子。

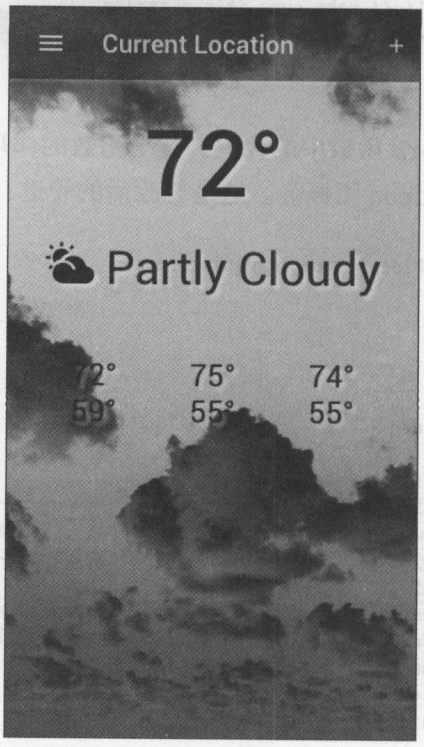



图 9-7：应用 Ionicon 天气图标后的 Ionic 天气 App

这个页面的样式终于搞定了。继续对侧滑菜单和位置页面进行样式化的工作就留给读者自己来完成了。

# 下一步

我们的天气 App 仍然还有一些需要改进的地方。最明显的是，我们自定义的城市无法保存。像城市列表这样的简单数据，如果用 Firebase 就有些杀鸡焉用牛刀了。你应该考虑用 local storage 或 Cordova File 插件来访问简单数据文件。具体哪一种就取决于你了。

还有一点，你可以尝试使用动态背景图。你可以用 Flickr API 根据位置或天气类型来抓取图片。

再有，你可以支持对温度显示单位的转换。

## 小结

通过这个 App，我们学习了如何使用侧滑菜单模板。我们还演示了通过管道将数据按制定模式进行转换。我们从外部资源加载了数据，学习了如何使用 Ionic 的代理机制解决 CORS（跨域资源共享）问题。最后分别使用 Ionic 事件和 Observable 机制来实现动态 UI 的更新。

# 第 10 章

# 调试并测试你的 Ionic 应用

终于到了要对你的 Ionic App 进行测试的时候了。在这一章，我们将教你如何利用各种方法和工具去解决可能出现的问题。因为 Ionic 是基于一系列技术构建的，所以 App 的调试也就不可能太简单。对于那些习惯于使用一个完整生态系统进行工作的开发者来说，这会是一个挑战。让我们来学习如何使用这些调试工具吧。

调试你的 Ionic App 可以分成几个不同的阶段进行：浏览器阶段、模拟器调试和真机调试阶段。每个阶段都提供了从多个角度的查看你的 App 的方式。因为 Ionic 是基于 Web 技术构建的，通常最好从浏览器开始进行调试。在切换到其他方法之前，我们其实还许多别的“第一步”要克服。

Ionic 2 使用的开发语言是 TypeScript，这是我们开发一个可以运行的 App 的第一道障碍。如果你的 IDE 在某些变量或函数上提示错误，这是提示某些错误的良好信号。有时是变量在重构过程中被不正确地修改了，有时是某个模块的路径少了一个子目录。发现这些小错误可以避免大量会导致你的 App 不正常的问题出现。

当然，你的 IDE 并不能在你敲代码时完美地发现任何问题。你可以用来解决问题的另外一个东西就是编译器的输出。你可能注意到了，在编译你的 Ionic 2 App 时，你的控制台窗口中会输出大量信息。如果你和大多数人一样，那么它可能只是一个卷缩在屏幕一角的小窗口。但是，我们不止一次从这个窗口看到有错误信息输出，只不过它们很快就滚动出我们的视线。

如果你的 App 因为某种原因无法启动，首先请看一下控制台中的编译信息。你可能会发现在一个 object 中少写了一个逗号或者代码中存在不容易发现的输入错误。我们建议调整你的控制台窗口的高度以便多显示几行代码。这就使你在问题出现时，能够一眼看到各种编译事件。

基于我们正在使用不完整的 IDE（和 Xcode 和 Android Studio 相反），我们无法获得某些编译调试工具的帮助。最常见的问题是，在你试图编译 App 时，会发现它无法识别出项目中存在未保存的文件。通常在我们编写某个页面时，可能会打开它的 HTML 文件来修改标签，打开 SCSS 文件来修改样式，同时打开 TypeScript 文件来编写组件代码。在下次编译之前，你很容易忘记保存其中的某个或某几个文件。这样在测试 App 时，你就会挠头了：奇怪，为什么有的东西会不对呢？因此在开始调试你的 App 之前，最基本的一个步骤就是确保所有的文件都已保存。

你有许多开发都是通过本地浏览器进行的，并使用 $ ionic serve 来运行 App 的。现在，ionic serve 有一个特性，就是监视所有源文件，一旦发现有文件被修改过，就会重新进行编译。这在许多时候都是非常好的，但 Ionic 的编译需要一定的时间。有时在编译脚本运行时，某些原因会导致在它们在前面的编译周期还未完成就被再次运行了。这会导致众所周知的“白屏死机”问题。实际上，这是因为 main.js 文件编译不正确或者干脆就不见了。如果你遇到这个问题，我们建议你先确认所有文件都已保存，然后再次运行 build。事实上，我们应该先停止 $ ionic serve，再重新执行。

如果你在 Chrome 中运行 App，Chrome 的标准开发者工具（DevTool）会成为你调试 App 的主要工具。你可能会记得我们曾经说过的一条开发原则：尽量将那些你不需要在真机上运行或测试的功能放在前面做。这能节省开发时间，利用 Chrome DevTools 快速开发出你的 App。如果你拥有传统的 Web 开发背景，这些工具的使用毋庸多说。但对于不太熟悉它们的人来说，我们建议你去 Google 的开发者网站（*https://developer.chrome.com/devtools*）读一读它的用法。

要打开 DevTools，你可以单击 Chrome 菜单，它就在浏览器窗口的右上角，然后选择“Tools”→“Developer Tools”。或者右键单击或 Ctrl+ 左键（Mac 系统），然后选择检查元素。

下面列出 DevTools 的一些常用快捷键：

- Ctrl+Shift+I（Mac 上是 cmd+opt+I），打开 DevTools。
- Ctrl+Shift+J（Mac 上是 cmd+opt+J），打开 DevTools 并将焦点置于控制台。
- Ctrl+Shift+C（Mac 上是 cmd+shift+C），以“检查元素”方式打开 DevTools，或者在 DevTools 已经打开的情况下，切换到“检查元素”模式。

通过这些工具，我们拥有了更多调试 Ionic App 的手段。如果有一个问题和某个元素的可视化外观相关，我们可以检查渲染后的 DOM 树来定位元素，然后用 CSS 检视器显示这个元素所使用的样式。这种手段通常会被我们用于理解某个元素的基本样式，这样我们可以在对 Sass 文件做最少的修改就可以实现所需的效果。

但是，我们用得最多的往往还是 JavaScript 控制台和配套的调试器。现在 Ionic 编译系统会将所有的源代码合并成一个文件，但通过源码映射文件，我们可以找到代码所在的最初位置。

在 DevTools 中有几个功能对手机开发者来说非常有用（见图 10-1）。

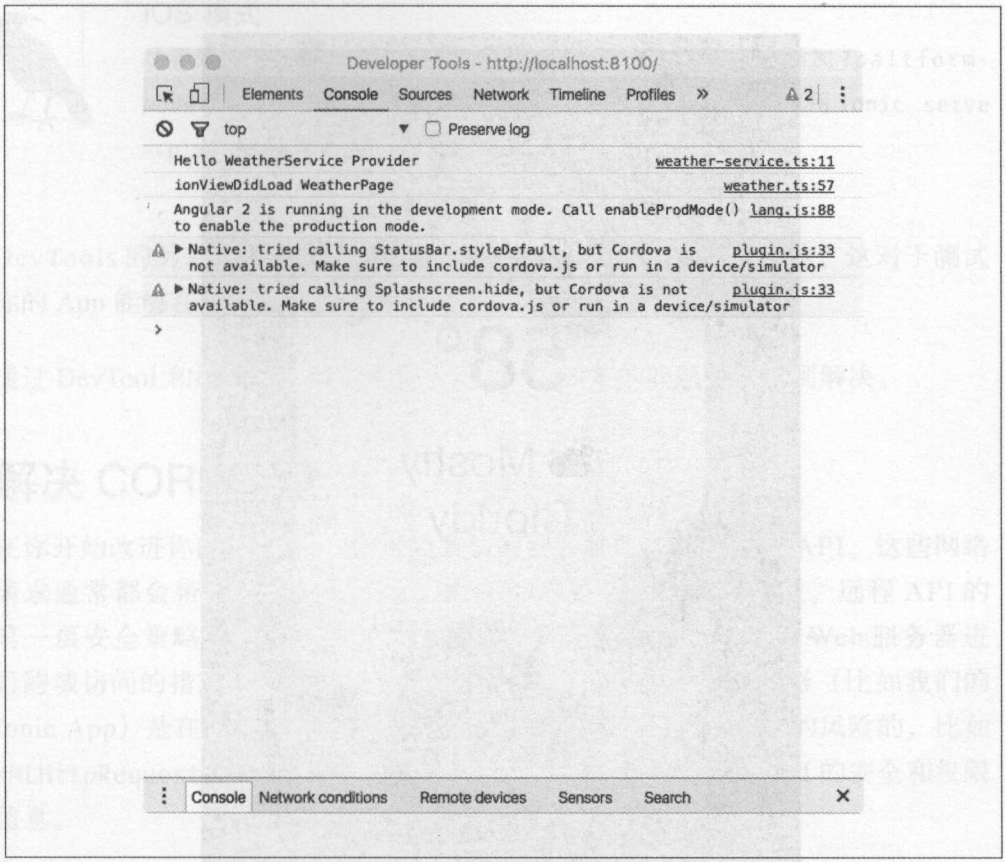



图 10-1：Chrome 的设备预览开关

在工具栏的左上角是设备按钮，它允许你改变浏览器窗口大小，以便你在手机屏幕上查看你的 Ionic App（见图 10-2）。


图 10-2：在 Chrome 设备模式下运行 Ionic 天气 App

这个按钮允许你选择各种主流的移动设备：exus 5x、Galaxy 5 和 iPhone 6，并且可以定制自定义的屏幕尺寸。对于某些设备，Chrome 会自动调整屏幕的外框（比如增加一个状态栏或者软按钮）。你还可以旋转屏幕，将屏幕从竖屏转变成横屏，并查看元素的改变。

**iOS 模式**

尽管列表中包含了 iOS 屏幕，你还是需要在 URL 地址后面添加 `?paltform-mode=ios` 才能让你的 App 适配 iOS 的显示。另外，你还可以用 `ionic serve --platform=ios` 来自动添加这个标志。

DevTools 的另外一个有用的地方是能够模拟各种网络条件和网速。这对于测试你的 App 能够在离线和在线两种情况下都能正确运行非常有用。

通过 DevTool 和浏览器，我们发现大约 70%~80% 的问题都能得到解决。

## 解决 CORS 问题

在你开始改进你的 App 时，你可能会访问到各种服务和它们的 API。这些网络请求通常都会带来一个问题，它们通常都需要某种类型的授权。远程 API 的第一层安全策略就是 CORS（跨域资源共享）。这是一种限制 Web 服务器进行跨域访问的措施，它保证了跨域数据的安全传输。现代浏览器（比如我们的 Ionic App）是在 API 容器中通过 CORS 来解决跨域 HTTP 访问的风险的，比如 `XMLHttpRequest` 或者 `Fetch`。请确认你知道你的 App 所用的 API 的安全和权限信息。

因为我们是在本地浏览器中测试我们的 App 的，这些外部请求都是被阻塞的。如果在模拟器或者真机上测试是没有这个问题的。有两种方法可以允许本地开发。你可以使用代理，就像我们在这个 IonicWeather App 所做的。还可以用另一个更快捷的方法，以禁用 CORS 模式打开 Chrome。当然这只能是用于测试目的。我们可以用命令行让 Chrome 以禁用 CORS 模式启动。对于 macOS 用户，这个命令是：

```
open -n -a /Applications/Google\ Chrome.app --args --user-data-dir=↵
"/tmp/chrome_dev_session" --disable-web-security
```

`-a` 标志用于打开指定的 App。`-n` 标志打开 App 的新实例，无论它是否已经运行。`--args` 后面的两个参数禁用了 CORS 并指定了用户数据目录。

对于 Windows 用户，则使用这个命令：

```
chrome.exe --user-data-dir="C:/Chrome dev session" --disable-web-security
```

*现在当你运行 `$ ionic serve`，Chrome 会运行在允许 HTTP 请求的模式下。*

# 用 iOS 或 Android 模拟器运行

除了用 DevTools 和本地浏览器之外，我们还可以怎样调试 Bug？答案是使用设备模拟器。尽管它无法完全模拟真正的设备，但它提供了最接近于真实设备的环境。为了以这种方式运行我们的 App，我们需要使用 `$ ionic emulate <platform>` 命令：

```
ionic emulate ios --consolelogs --serverlogs

ionic emulate android --consolelogs --serverlogs
```

或者简化为：

```
ionic run ios -c -s

ionic run android -c -s
```

### livereload 标志

Ionic CLI 通过 `--livereload` 或 `--l` 标志支持实时加载。但是由于某些底层的安全保护方面的原因使这个特性不能正常工作。Ionic 团队正在努力解决这个问题。

尽管我们是在模拟器中运行 App，我们仍然有一些其他的强大的调试工具可供选择。对于 macOS 用户，Safari 将是我们找到和解决 Bug 的一个工具。选择“Safari”→“偏好设置”→“高级”，在菜单栏中勾选“显示开发菜单”。

在新增加的开发菜单中，我们可以在“Simulator”中选中正在运行的我们的 App，然后对它进行调试。Safari 的开发者菜单和 Chrome 的 DevTools 很像，它们都能够查看页面的 DOM 结构，源文件和 JavaScript 控制台。

## Android

Chrome 也提供了类似的功能。在地址栏中，输入 **`chrome://inspect`**。这会显示一个 Chrome 能够进行远程调试的设备列表。当你的 App 在模拟器中启动成

功，它就会在这个列表中显示出来。现在你可以用 Chrome DevTools 来调试在 Android 模拟器中运行的 App 了。

## 在设备上进行调试

就像模拟器测试要在浏览器测试的后面进行一样，最后一个测试步骤是在真机上进行。要在真机上进行测试，我们需要编译我们的 Ionic App 并安装到真机上。为了让我们的 Ionic App 运行在这种模式，我们必须使用 `$ ionic run <platform>` 命令：

```
ionic run ios

ionic run android
```

这会生成一个能够在真机上加载并运行的 IPA 文件（iOS）或 APK 文件（Android）。我们针对这两个平台分开讨论。

### Android

在设备上安装我们的 App 之前，需要打开 Android 的设备调试选项。这个选项位于开发者选项设置中，默认它是不显示的。要显示这个设置项，需要打开设置，进入关于本机一节。然后在“Build Number”上连续单击七下。如果你无法找到这个选项，你可以用搜索引擎找到针对你的设备的特殊方法。解锁了这个选项后，你就可以打开“允许 USB 调试”选项了。

这里有一个限制：要使用 Chrome 的远程调试功能，设备必须运行在 Android4.4（KitKat）或以上（也就是 API 级别 19）。同时，需要电脑上安装 Chrome 版本 30 及以上。

要在你的 Android 设备上加载 App 有多种方式：将 APK 上传到一台 Web 服务器上，将它拷贝到类似 Dropbox 这样的服务上，或者用 email 发送这个 APK。当你从设备上下载这个 APK，Android 安装器就会运行，提示 App 需要的权限然后开始安装 App。

和你在 Android 模拟器中调试 Ionic App 一样，在真机上调试 App 的步骤也是一样的。打开 Chrome 输入 *chrome://inspect*。将你的 Android 设备插到电脑上，当

Ionic App 运行，你会看到你的 App 会列出来。选择你的 App，然后使用我们熟悉的 Chrome DevTools。

如果你能够在检查设备中看到你的设备，但不能看到 Cordova Web View，你需要在 AndroidManifest.xml 文件的 <application> 节点中添加 `android:debuggable= "true"`。

现在，我们已经完全可以和运行在原生环境中的 App 进行交互了。例如，你的 App 可以调用设备上的相机，而这个功能在浏览器和模拟器上都是无效的。

### iOS

在 iOS 上调试你的 Ionic App 和 Android 非常类似。

但有一点，你仍然需要对 IPA 用开发者证书进行签名，并确认你在 *config.xml* 中的 App ID 和开发者证书中的完全一致。如果不一致，你的 IPA 文件无法正确签名，也无法在 iOS 设备上运行。

如果你在用 Ionic CLI 生成 IPA 时出现错误，可能需要打开对应的 Xcode 项目，手动设置签名的 profile。一般设置完成后 Ionic CLI 就会正常。

当 IPA 创建并安装到你的 iOS 设备上之后，将你的 iOS 设备连上电脑，打开电脑上的 Safari。然后，启动 iOS 设备上的 Ionic App。在 Safari 开发菜单中，你会看到你的 iOS 设备列出。可能需要等一小会 WebView 才会列出。选中它，你就可以和在模拟器上测试一样使用开发者工具了。

## 调试 Ionic 的初始化

有时，在真机或者模拟器上测试 Ionic App 时会在 App 初始化过程中出现问题。正常情况下，在远程调试器找到并连接好 App 之前，你的 App 必须运行。这样，有一些控制台信息或错误是无法被捕获的。当你的 App 和远程调试进程连上之后，我们才能够看到我们错过的那些错误。你可以用 Dev 工具栏中的 Reload 按钮或者在控制台中输入这条命令：

```
window.location.reload();
```

这会强制 WebView 重新加载页面，因为这个时候远程调试器已经连接，你就能够看到那条错误是什么了。

## 其他工具

除了这些调试工具，还有几个别的工具在 Ionic 开发中也会用到：

*Visual Studio Code 的 Cordova Tools Extension*

当使用 Visual Studio Code 的时候，Cordova Tools Extension 提供了代码提示、调试和集成 Apache Cordova（PhoneGap）CLI 的功能。特别值得一提的是，它还支持 Ionic 框架（虽然只是版本 1）。在 Ionic 开发过程中，我们发现这个扩展非常有用。

*Tools for Apache Cordova (TACO)*

如果你进行 Ionic 开发的 IDE 是 Visual Studio，你还可以安装 Tools for Apache Cordova，简称 TACO。这个扩展提供了智能感知、调试、编译 Apache Cordova 和 Ionic 项目的支持，而且和 Corova Tools Extension 不同，TACO beta 版支持 Ionic 2。像 Apache Ripple 一样，Visual Studio 也可以调试运行在 iOS、Android、Windows 设备、模拟器或虚拟机、浏览器上的代码。

TACO 还有一个功能，它能够使用 MacInCloud 类似的服务编译 Mac OS 平台。

*GapDebug*

Genuitec 的 GapDebug（*https://www.genuitec.com/products/gapdebug/*）是另一个在我们的工作中行之有效的调试方案。它支持 Windows 和 Mac 平台，这个工具支持以简单拖曳的方式将 App 文件安装到你的设备上。它集成了 Safari 的 Web Inspector（iOS）和 Chrome DevTools（Android）。无论在 Windows 还是 macOS 上都可以对 iOS 和 Android App 进行调试。不过，Genuitec 最近宣称终止对该产品的开发。

*Ripple*

Ripple 是一个针对 Cordova 项目的桌面模拟器。实际上，它能够在你的桌面 App 中运行一个 Cordova App，模拟各种 Cordova 特性。例如，它能

够模拟加速计以测试摇晃事件。它通过让你从硬盘上选择图片来模拟相机的 API。Ripple 让你将精力集中于特定的地方，以便你不需要过多地操心 Cordova 插件。

*Augury*

Augury 正式名称是 Batarangel，是一个 Chrome 开发者工具的扩展，允许开发者查看他们的 Angular 2 App 组件树和相关的数据。这个方案几乎是和 Ionic 2 的成长一起出现的。

# 小结

如你所见，调试 Ionic App 的方法因为测试环境的不同而不同。根据开发阶段的不同，要解决可能出现的问题，应当正确选择所要使用的工具集。

第 11 章

# 部署你的应用程序

现在，你的 App 已经经过了充分的测试，接下来该构建用于提交到各个应用商店的版本了。在我们编译 App 的 release 版本之前，我们应该将不需要包含到 release 版中的 assets 移除。

例如 Console 插件，这个插件是我们在第一次编译我们的 Ionic App 时添加的，我们应该将它从已安装的插件中删除掉。这只需要执行命令：

```
$ ionic plugin remove cordova-plugin-console
```

我们还需要在 config.xml 中手动删除这个插件的引用。找到这一句，将它删除：

```
<plugin name="cordova-plugin-console" spec="~1.0.4"/>
```

## 修改 config.xml 文件

在编译 release 版本之前，你必须修改 *config.xml* 文件中的某些设置。默认，Ionic 生成的 *config.xml* 十分简单。你必须为 App 添加更多的设置，比如限制 App 能够运行的 OS 版本。在 *config.xml* 中有些设置是可以修改的，比如版本号、限制能够访问的域名列表、将开发时候使用的 API 端点替换为生产所用的 API 端点等。

你可以参考附录 B 中关于这些设置项的简要说明，还可以参考 Cordova 网站的完整文档（*http://bit.ly/2l8zSqd*）。

## App 图标和 splash 图片

如果你没有将 Ionic 模板提供的 App 图标和启动画面替换掉，那么现在就是做这个事情的时候了。不需要为每个平台导出一大堆不同尺寸的图片，我们可以用 Ionic CLI 来自动生成这些图标。

首先，删除现有的图标和启动图片。我们发现 CLI 不会覆盖已经存在的文件，它只会不声不响地出错。然后，在 resources 文件夹中添加你的基本图标文件。为了获得最佳效果，这个文件必须是 1024 × 1024 像素。不要为某个平台添加特殊效果，比如圆角、加亮等。准备好图标文件之后，将它命名为 icon.png。CLI 也支持 Photoshop 文件（.psd），但为了防止你使用了某些不支持的滤镜或特效，我们建议你使用一张“弄平后的”的 png 文件。

对于启动图片，你可以使用 2208 × 2208 像素的文件。和四方形的图标有所不同，启动图片可以是长方形的，包括横屏的和竖屏的。如果你的 App 只支持一个方向，你只需要提供一个方向的启动图片。Ionic 团队提供了一个模板（*http://bit.ly/2mXe6Fc*），用于说明你的图片中那一部分是安全区。将文件保存为 splash.png，也放到 resources 目录。

在命令行中，运行 `$ ionic resources`。这会将文件上传到 Ionic 以生成各种大小的图标，然后再将它们保存到合适的目录里。

## 编译你的 Android APK

要构建 Android release 版，我们可以使用这个 CLI 命令：

```
$ ionic build android --release -prod
```

如果你曾经编译过 Ionic 1 App，你可能发现了一个新的标志 `-prod`。这个标志告诉 Build 脚本在编译 *main.js* 时使用 *main.prod.ts* 文件代替。对 Angular 2 开发熟悉的读者，应该知道这个文件会调用 `enableProdMode()` 方法。

当这句 CLI 命令执行完毕（但愿没有出现错误），我们会在这里找到我们的没有签名的 APK 文件：*platforms/android/build/outputs/apk/android-release-unsigned.apk*。现在，我们需要对这个 APK 进行签名，并使用某些对齐工具进行优化，并准备提交应用商店。

## 生成签名密钥

要生成签名密钥，我们需要用到 keytool 命令，这个命令在安装 JDK 的时候就已经有了。这是这个命令的基本用法：

```
$ keytool -genkey -v -keystore my-release-key.keystore -alias alias_name
-keyalg RSA -keysize 2048 -validity 10000
```

*alias_name* 是你的密钥的别称。这只是一个描述性的文字，用于识别你的 App。它可以是字母、数字和下划线。注意密钥的别名是大小写敏感的。你可以将 *my-release-key.keystore* 修改为别的什么。

这个工具会提示你为 keystore 文件和密钥别名输入一个密码。然后会在生成密钥的过程中问你一系列问题：

```
Enter keystore password:
Re-enter new password:
What is your first and last name?
  [Unknown]:  Chris Griffith
What is the name of your organizational unit?
  [Unknown]:  None
What is the name of your organization?
  [Unknown]:  AJ Software
What is the name of your City or Locality?
  [Unknown]:  San Diego
What is the name of your State or Province?
  [Unknown]:  CA
What is the two-letter country code for this unit?
  [Unknown]:  US
Is CN=Chris Griffith, OU=None, O=AJ Software, L=San Diego, ST=CA, C=US correct?
  [no]: y
```

回答完这些问题，你会在终端窗口中看见：

```
Generating 2,048 bit RSA key pair and self-signed certificate (SHA256withRSA)
with a validity of 90 for: CN=Chris Griffith, OU=None, O=AJ Software,
L=San Diego, ST=CA, C=US

Enter key password for <my_alias_name>
    (RETURN if same as keystore password):
```

当这一切完成，你可以在当前命令的工作目录下看到一个 *my-release-key.keystore* 文件。

将你的 keystore 文件保存在一个安全的地方。当你用这个 keystore 签名并提交

你的 App 时，所有对这个 App 的更新都必须用相同的 keystore 进行签名。没有任何方法重新生成另外一个 keystore 文件。

然后，我们开始对 APK 进行签名。要进行签名，我们需要用到 jarsigner 工具，这个工具也是 JDK 中自带的：

```
$ jarsigner -verbose -sigalg SHA1withRSA -digestalg SHA1
-keystore my-release-key.keystore android-release-unsigned.apk alias_name
```

这个命令会询问你创建 keystore 时创建的密码，然后对 App 进行签名。

最后一步是使用 `zip align` 工具（位于 */path/to/Android/sdk/build-tools/VERSION/zipalign* 目录）优化 APK 包：

```
$ zipalign -v 4 path/to/android-release-unsigned.apk MyApp.apk
```

这样，我们就创建了一个名为 *MyApp.apk* 的 release APK，可以用于提交给 Google Play 商店了。

## 提交 Google Play 商店

现在我们来讨论真正提交一个 App 到 Google Play 商店的基本步骤。首先，你需要创建一个 Google 开发者账号（*https://play.google.com/apps/publish/*）。这个账号需要一次性付费 25 美金。

创建好开发者账号，你需要登录到 portal，开始发布过程，这一步非常简单。当然，如果你已经准备好所有 Play 商店列出的必需的相关 assets 的话，则会更简单，比如截屏和销售信息（见图 11-1）。

提交完所有材料，你的 App 将在最多几个小时内即可上架销售。

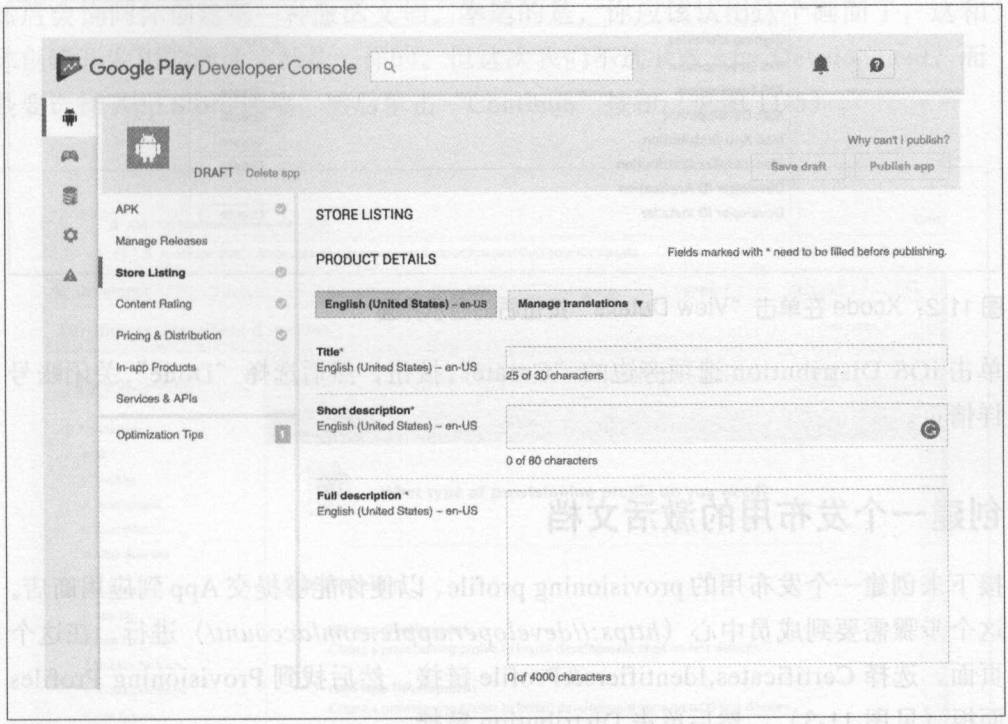



图 11-1：Google Play 商店 App 提交界面

# 编译你的 iOS App

尽管我们可以不用注册苹果开发者计划（*https://developer.apple.com/programs/*）也能够编译 iOS App 的开发版本，但如果你想将它发布到苹果的 App 商店，你必须支付每年 99 美金的费用。当你注册完成后，我们必须配置 Xcode 以使用这个账号。在 Xcode 中，打开 Preferences → Accounts，填入你的苹果 iOS 开发者账号信息。

## 请求发布证书

打开 Xcode，登录你的开发者账号，打开“Preferences”→“Accounts”，选择你的苹果 ID，单击“View Details”按钮。你会看到一个如图 11-2 所示的窗口弹出。

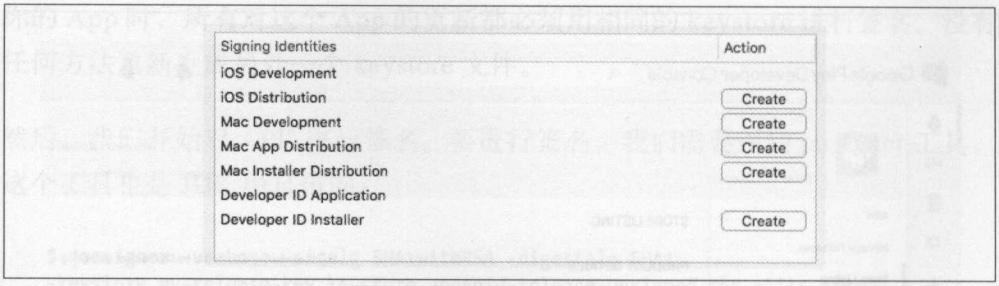



图 11-2：Xcode 在单击“View Details”按钮后的显示界面

单击 iOS Distribution 选项旁边的“Create”按钮，然后选择“Done”关闭账号详情页。

## 创建一个发布用的激活文档

接下来创建一个发布用的 provisioning profile，以便你能够提交 App 到应用商店。这个步骤需要到成员中心（*https://developer.apple.com/account/*）进行。在这个页面，选择 Certificates,Identifiers&Profile 链接。然后找到 Provisioning Profiles 面板（见图 11-3），然后单击 Distribution 链接。

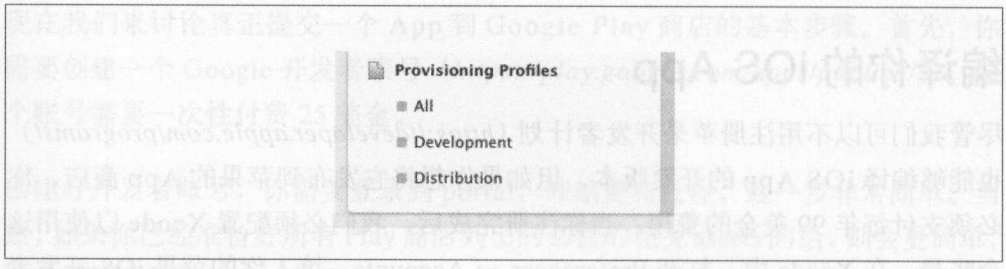



图 11-3：Provisioning Profiles 面板

在这个页面，单击 [+] 按钮，开始生成发布用的激活文档（见图 11-4）。

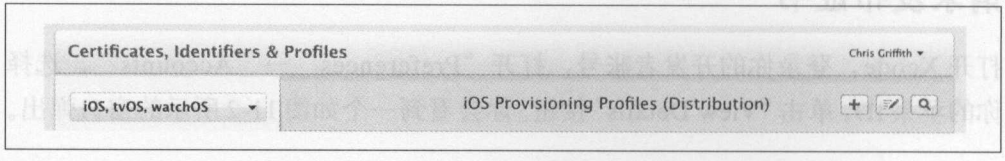



图 11-4：单击 Provisioning Profiles 面板中的 [+] 按钮

然后会询问你创建哪一种激活文档。幸运的是，你应该认出这个画面了，这和你创建开发用的激活文档是一样的。但这次我们不选 iOS App Development，而是要选择 App Store 选项，然后单击“Continue”按钮（见图 11-5）。

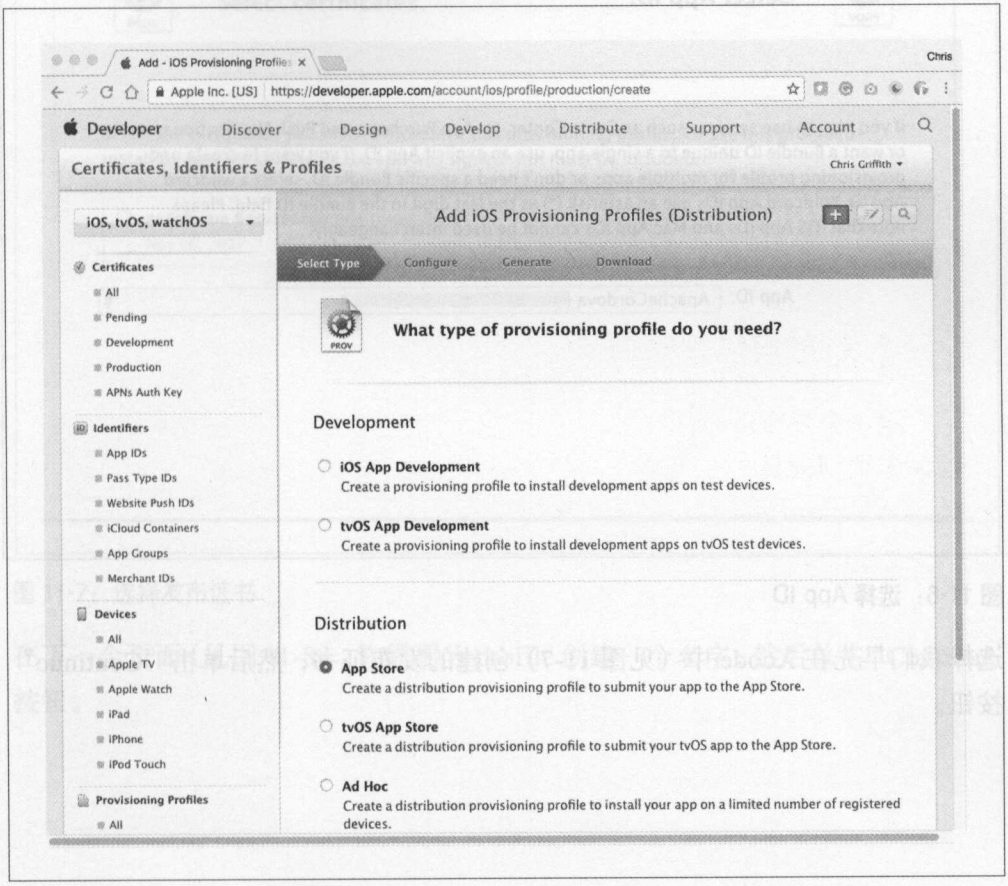



图 11-5：选择 provisioning profile 的类型

在 App ID 选择页面（见图 11-6），我们需要选择和该 App 所对应的 App ID。在创建开发证书时，你应该生成了一个 App ID。如果还没有，请退出这个界面，单击 Identifiers 面板中的 App ID 链接生成一个。一旦你选择了你的 App ID，单击“Continue”按钮进入下一步。

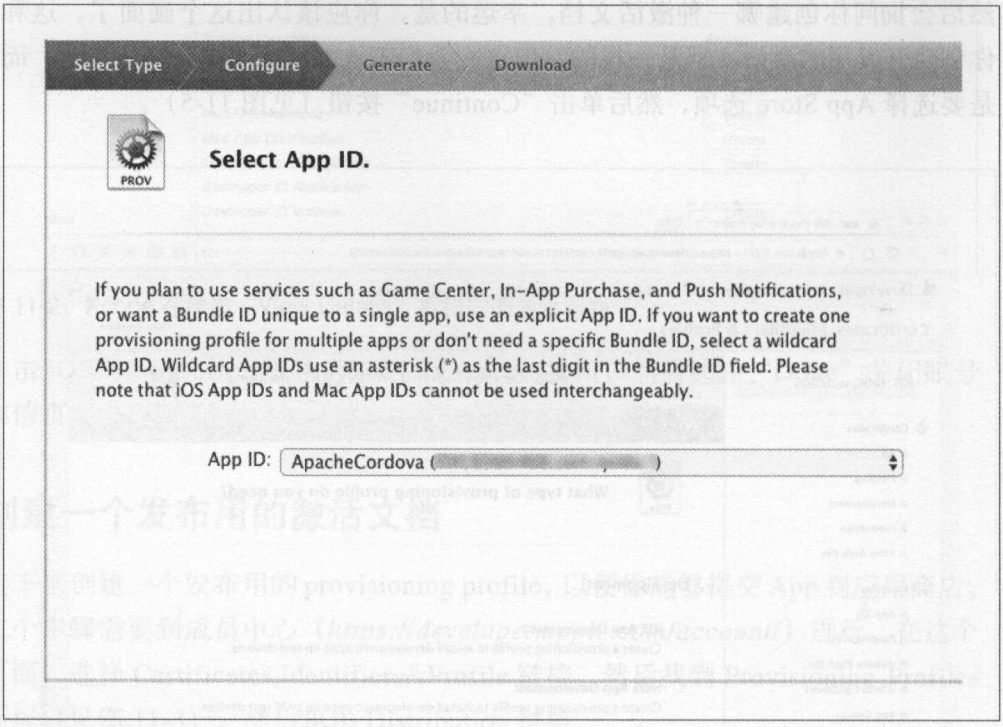



图 11-6：选择 App ID

选择我们早先在 Xcode 中（见图 11-7）创建的发布证书，然后单击“Continue”按钮。

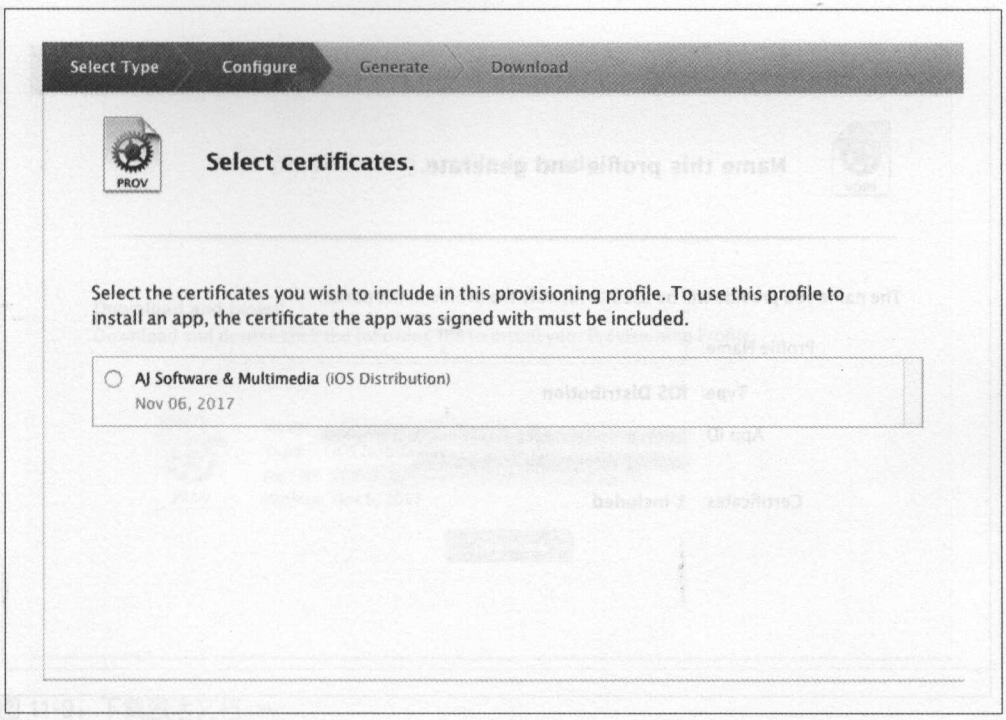



图 11-7：选择发布证书

在下一个页面（见图 11-8），你需要为 profile 创建一个名字，然后单击“Continue”按钮。

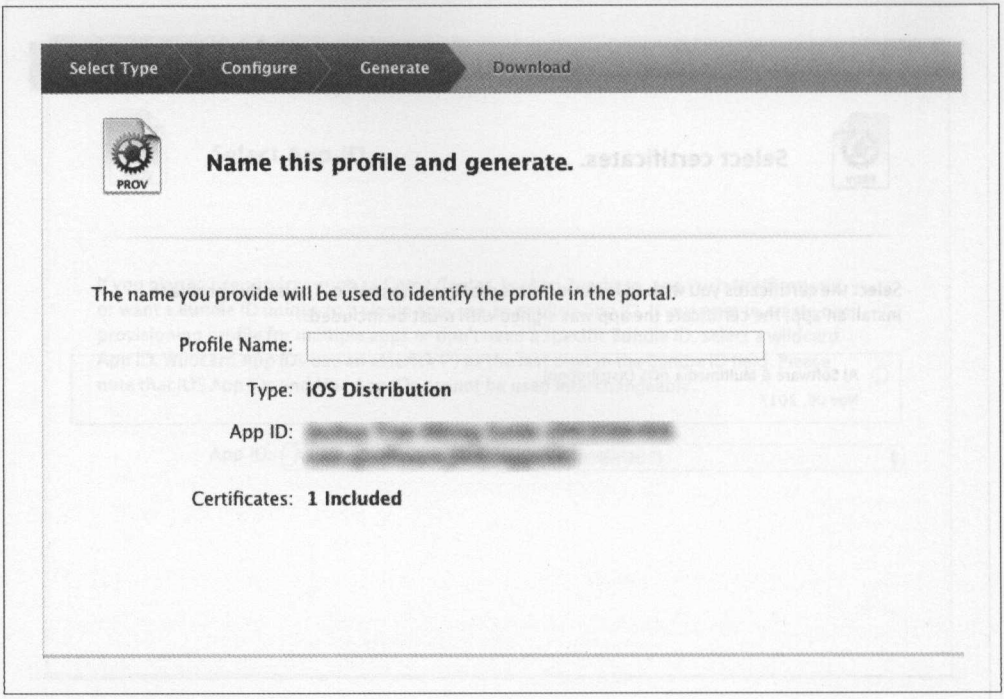



图 11-8：创建 Profile

这会创建激活文档，单击“Download”按钮（见图 11-9），将它下载到你的电脑。

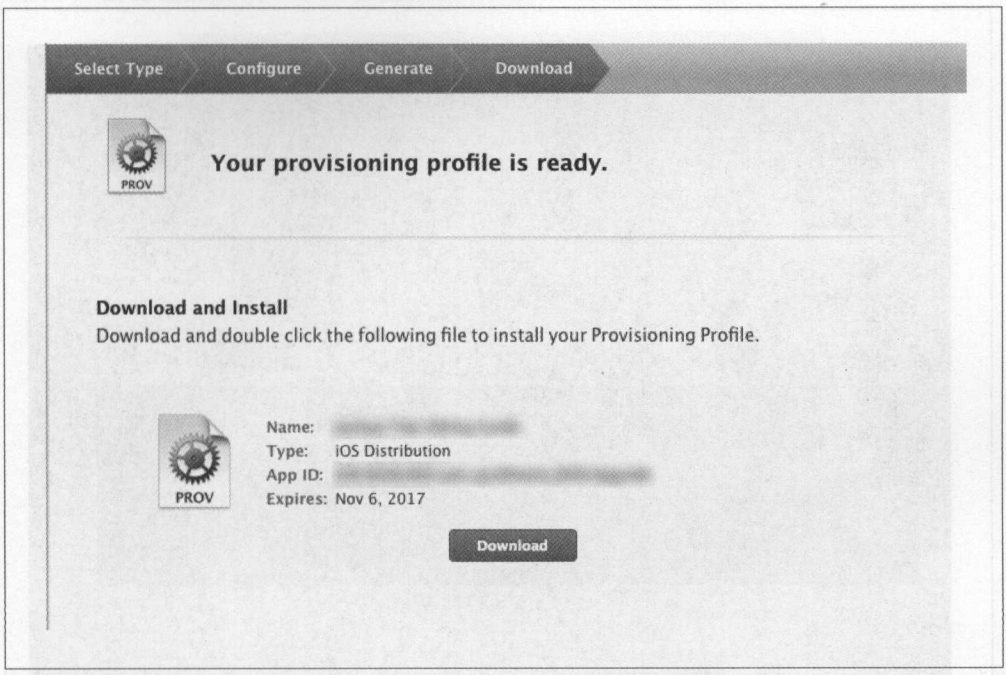



图 11-9：下载激活文档

一旦你下载好激活文档，双击它进行安装。

## 创建 App 列表

苹果使用 iTunes Connect（*https://itunesconnect.apple.com/*）来管理 App 的提交。当你登录进去之后，你会看到如图 11-10 所示的界面。

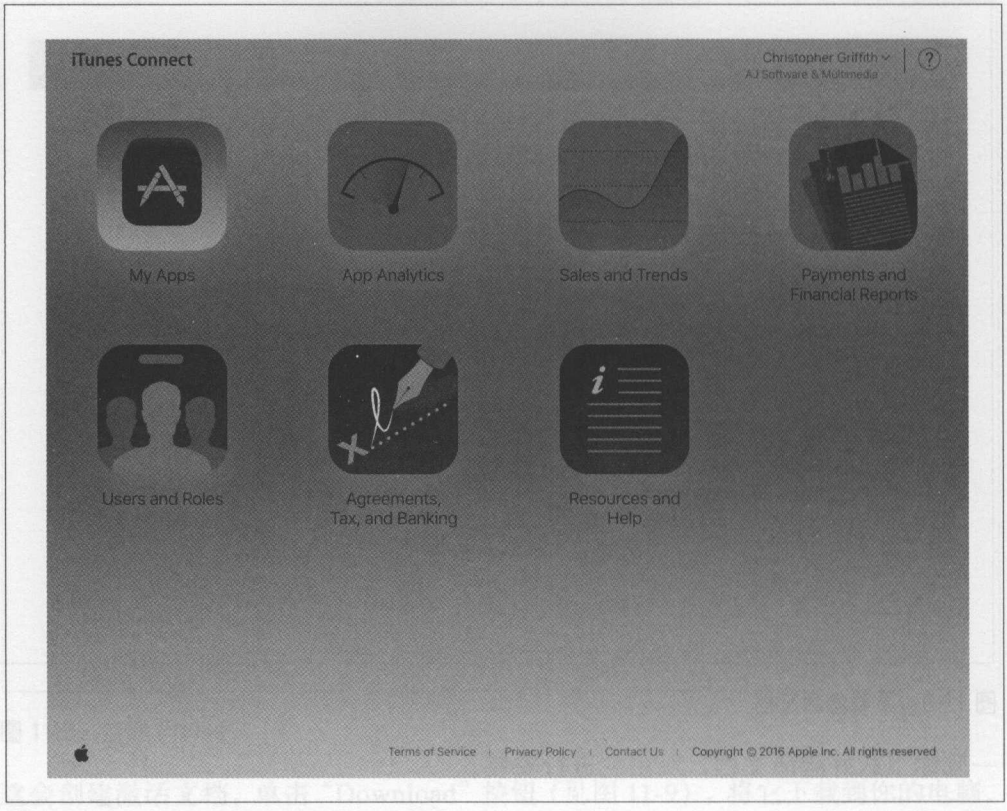



图 11-10：iTunes Connect 门户

选择“My Apps”按钮，然后单击左上角的 [+] 图标。这会打开如图 11-11 所示的对话框。


图 11-11：新 App 对话框

选择 iOS 平台，然后填写剩余信息。填写完成后，单击“Create”按钮。

App Information 页面会显示（见图 11-12）。虽然在这里我们也可以设置一些信息，但我们还是稍后在回到这个页面。要创建一个 placeholder App 的原因是 App 的信息必须在 Application Loader 工具上传 App 之前就已经存在。

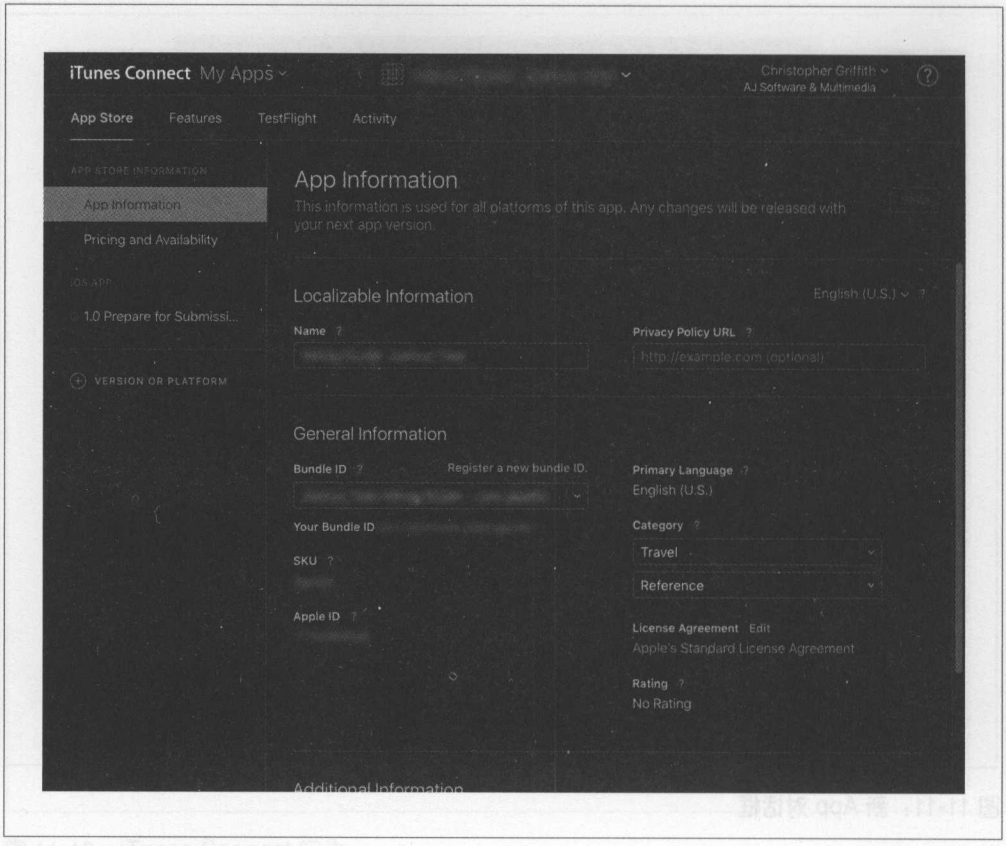



图 11-12：App 基本信息页面

## 编译用于生产的 App

当我们的“假的”App 创建之后，我们就可以继续进行发布编译了。返回命令提示窗口，运行 `$ ionic build ios --release -prod`。终端窗口会输出了大量信息，几分钟后，你应该在控制台中看到 `BUILD SUCCEED` 信息。否则，请拖曳滚动条查看具体的错误信息。

## 创建 App 的 Archive

要提交我们的 App 到 iTunes Connect，我们必须创建 App 的 Archive。回到 Xcode，打开你的 App 的 Xcode 项目文件，这个文件应该位于 *platforms/ios* 目录。

在项目信息窗口，检查 bundle ID 是否正确，以及版本号和 build 数正确。team 选项应当设置为你的苹果开发者账号。如果你想对运行 App 的设备种类进行指定，也可以在这里进行修改它。

然后选择菜单“Product”→“Scheme”→“Edit Scheme”，打开 scheme 编辑器。在列表中，选择“Archive”。确认 Build Configuration 已经设置为 Release（见图 11-13）。

要创建 Archive，我们需要确认工具栏中的设备列表的 target 被设置为 Generic iOS Device。然后，单击菜单“Product”→“Archive”，Archive organizer 窗口弹出，显示 new archive 界面。

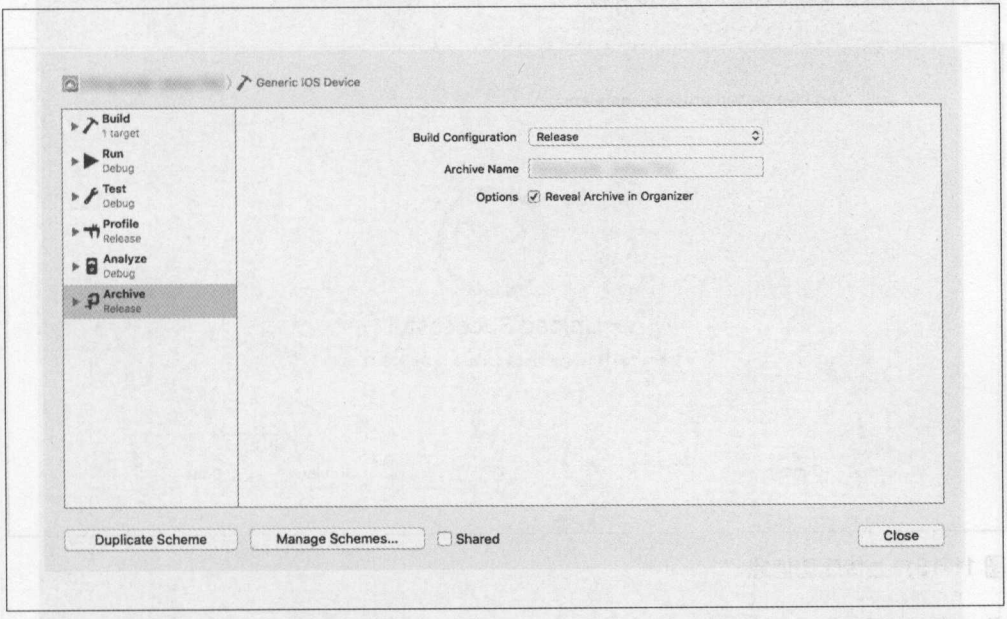



图 11-13：Build 设置窗口

我们的 App 现在可以上传到 iTunes Connect 了。单击“Upload to App Store”按钮（见图 11-14）。Xcode 会签名我们的 App，对它进行校验，然后上传到 iTunes Connect。当上传完成，你会看到如图 11-15 所示的画面。

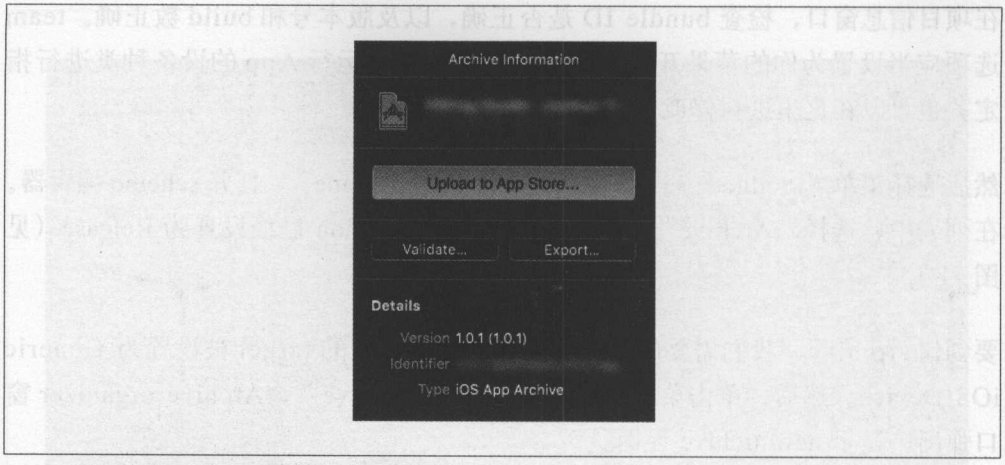



图 11-14：App Store Archive 上传面板

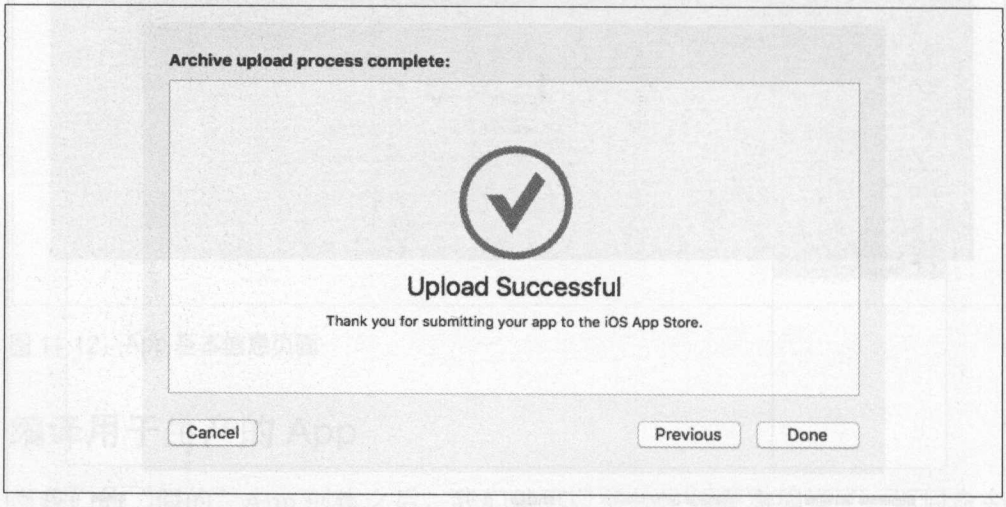



图 11-15：上传成功信息

几分钟之后，你会收到确认邮件。我们可以回到 iTunes Connect 门户的 App 列表，完成提交过程。

这里，你有两种提交方法：你可以将 release App 发布到商店审核并上架，也可以发布到 TestFlight 用于测试。我们简单地看一下这两种方法。

## 用 TestFlight 进行 beta 测试

TestFlight 测试是苹果提供的一个服务，允许你邀请用户测试你的 iOS App，然

后再将它发布到应用商店。当你将 iOS App 上传到 iTunes Connect 门户之后，在开始的第一步单击 TestFlight 标签。你需要填写一些基本信息：一个 email 地址用于搜集反馈信息，一个用于市场的 URL，以及一个隐私策略声明。

然后，你需要回答几个关于加密的问题。你的 App 会经过苹果的一个短暂的评审过程。当评审通过，你会收到邮件通知，你就可以将它发给内部 / 外部的测试人员。你的测试人员需要安装一个 TestFlight App 才能访问你的 App（见图 11-16）。

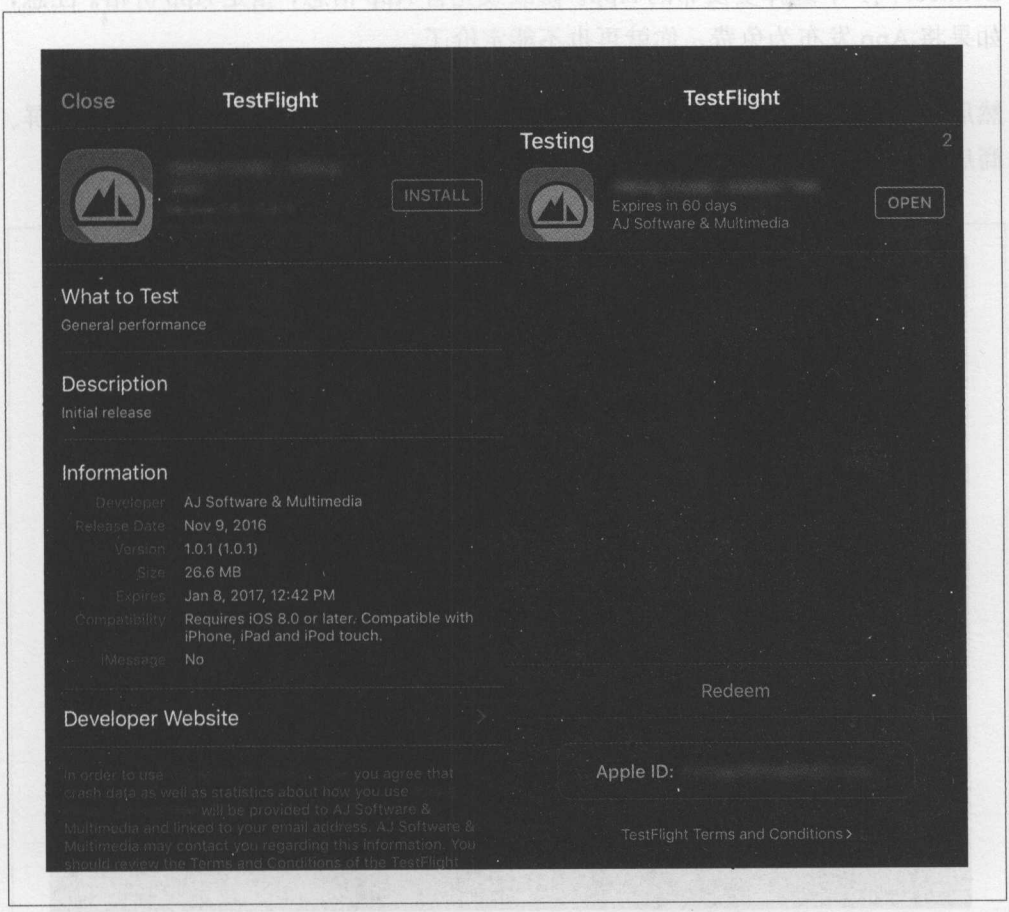



图 11-16：TestFlight App

当你准备发布测试 App 时，你的测试人员会收到一封 email，内容中包含了一个链接，这个链接会打开他们设备上安装的 TestFlight 并安装该 App。然后他们就

可以像使用其他 App 一样使用这个 App 了。有一点需要注意：这些 App 发布之后会有 60 天的有效期，因此它不可替代长期的发布方案。测试人员数一次不能超过 2000 位。当测试结束，你可以将它们转移到 App Store 中进行发布。

## 发布到应用商店

当我们的 iOS App 上传并校验完成，最终需要将它提交给苹果审核。在 iTunes Connect 门户，选择要发布的 App。修改或完善 App 信息，指定 App 价格。注意，如果将 App 发布为免费，你就再也不能定价了。

然后单击 Prepare for Submission 链接。这会显示一个长长的表单，包括 App 截屏、商店描述、关键字和 App 分级等（见图 11-17）。

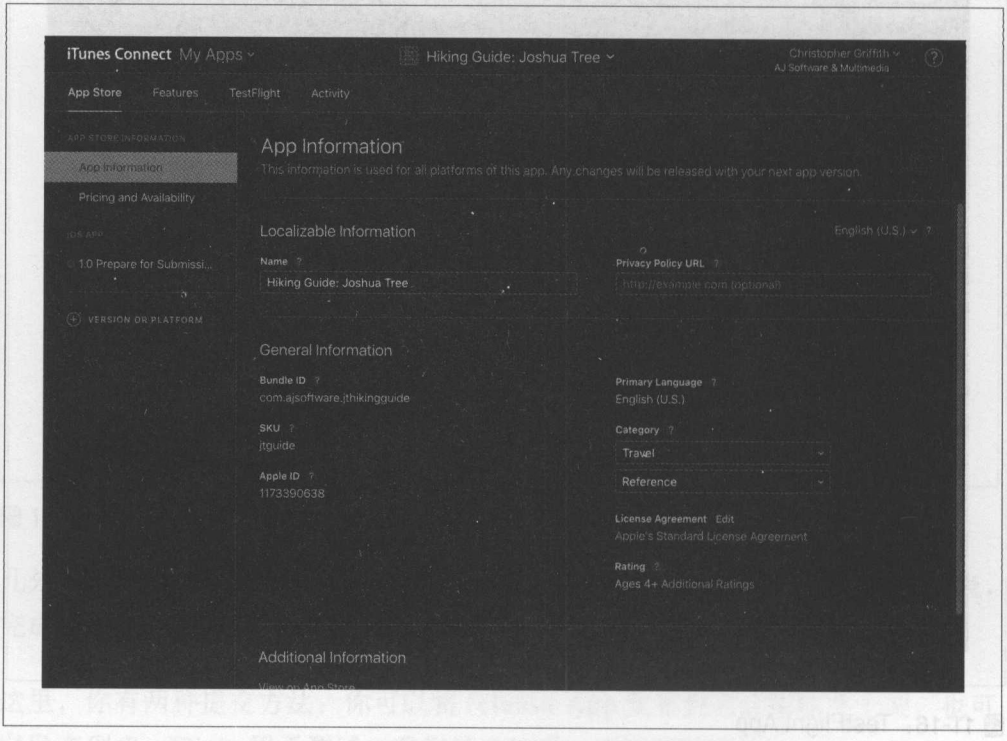



图 11-17：iTunes Connect App 详情页面

一旦你编辑完这些内容，你就可以将 App 提交给苹果评审。这个过程可能是几天或者更长，取决于 App 评审工作量的多寡。现在，你可以将销售新 App 的艰巨任务提到日程上了！

## 小结

你现在已经知道如何正确编译并提交 App 到苹果应用商店和谷歌 Play 商店了。如果你准备发布到 Windows 商店，微软的 TACO 团队提供了一个很好的流程说明（*https://taco.visualstudio.com/en-us/docs/tutorial-package-publish-readme/#package-the-windows-version-of-your-app*）。

第 12 章

# 探索 Ionic Cloud

Ionic 不仅仅是一个用于创建移动应用的框架，同时也是一个云服务的集合。这些服务包括能够远程编译 App、能够不经过二次提交即可更新已发布的 App，以及通过可视化的编译器生成用户界面。Ionic Cloud 的目的是真正支持对移动应用的部署。和开源的 Ionic 框架不同，这些服务需要按月付费。但不用担心：其中大部分都会提供开发版，你可以免费试用它们，只是使用上有所限制。

最初的时候 Ionic Cloud 还包含一些其他服务，比如用户认证、推送通知、统计分析和数据库支持。这些服务都已经或即将关闭。

## 创建 Ionic Cloud 账号

在能够使用任何 Ionic Cloud 服务之前，我们必须创建一个免费的云账号（*https://apps.ionic.io/signup*）。

然后我们需要将 Ionic Cloud 客户端安装到 App 中。确认你的当前目录位于 App 的根目录下。然后运行安装客户端的命令：

```
$ npm install @ionic/cloud-angular --save
```

# 生成你的 Ionic App ID

在为我们的 App 配置 Ionic Cloud 之前，我们必须拥有一个 Ionic App ID。这个 ID 和 *config.xml* 文件中的 App ID 不是一回事。仍然在你的项目目录下，运行这个命令：

```
$ ionic io init
```

这个命令会自动生成你的 Ionic App ID 并保存在你的 *ionic.config.json* 文件中。你可能需要输入你的 Ionic.io 账号和密码。当 *ionic.config.json* 文件处理完，直接打开这个文件，其中的一些东西在我们的下一个步骤中会用到。

除了更新我们的本地 App，这个命令还会在你的 Ionic Cloud Dashboard 中创建一个东西（见图 12-1）。

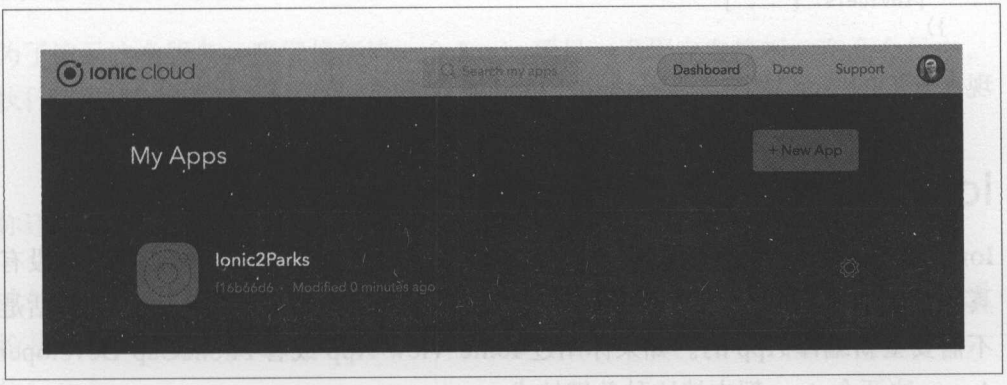



图 12-1：Ionic Cloud Dashboard

要访问 Ionic Cloud Dashboard，请登录 Ionic.io。

# 配置你的 App

一旦向 Ionic Cloud 注册了我们的 App，就可以修改 App 的 bootstraping 了。在编辑器中打开你的 *app.module.ts* 文件。

首先需要导入 Ionic Cloud 模块：

```
import { CloudSettings, CloudModule } from '@ionic/cloud-angular';
```

接着，声明一个常量，用于定义我们的 Ionic Cloud 相关设置。将下面代码中的 APP_ID 替换成你的 *ionic.config.json* 文件中的对应值：

```
const cloudSettings: CloudSettings = {
  'core': {
    'app_id': 'APP_ID'
  }
};
```

在 @NgModule 中，我们必须向 imports 数组中引入我们的 CloudModule 模块：

```
@NgModule({
  declarations: [ ... ],
  imports: [
    IonicModule.forRoot(MyApp),
    CloudModule.forRoot(cloudSettings)
  ],
  bootstrap: [IonicApp],
  entryComponents: [ ... ],
  providers: [ ... ]
})
```

现在我们的 App 就可以调用 Ionic Cloud 服务了。

## Ionic 部署

Ionic Cloud 中提供了一个 Ionic Deploy 服务。因为我们 App 的核心代码并没有真的被编译成本地代码，所以如果仅仅是替换我们的 HTML、JS 和 CSS 的话是不需要重新编译 App 的。如果你用过 Ionic View App 或者 PhoneGap Developer App，这两个 App 都支持这种关键技术。

你的 App 一旦使用了实时发布，你就能够：

- 跳过苹果的评审。
- 随时更新 App。
- 毫不费力地将新功能推送给用户和进行 bug 修复。
- 进行 A/B 用户测试。

在使用 Ionic Deploy 时，你需要了解以下专有名词：

*快照*

你的 App 的源代码的打包后的版本。这个包会发布给用户设备。

*实时发布*

即对应的快照可以下载并运行到设备上。

*通道*

快照被发布到某个发布通道。通道允许你将不同 App 快照发布到不同设备上。

*二进制更新*

当你需要更新 App 的原生代码时，通常需要更新 Cordova 或 Cordova 插件。

## 创建 Ionic 部署

为了演示这个服务，我们将新建一个 Ionic 项目，采用空白模板。在命令行中，执行命令：

```
$ ionic start IonicDeploy blank --v2
```

你还要安装基本的 Ionic Cloud 组件并配置你的 App 以使用 Ionic Cloud。

此外，你还要安装 Ionic Deploy 插件。这个插件用于负责 App 的实际更新，无论是 Android 设备还是 iOS 设备。在终端窗口中，切换到你的 App 的项目目录，执行命令：

```
$ ionic plugin add ionic-plugin-deploy --save
```

安装好这个插件，我们就可以检查、下载和抽取新包，然后使用它们。

## 测试 Ionic 部署

Ionic 部署功能只能在模拟器或真机上运行。你无法在浏览器上使用这个服务。

### 启用 Ionic 部署

打开 *home.ts* 文件，在 import 语句中添加 Deploy 模块：

```
import { Deploy } from '@ionic/cloud-angular';
```

然后在构造函数中加入它：

```
constructor(public navCtrl: NavController, public deploy:Deploy) {

}
```

在继续下一步之前，我们先来了解一下发布通道的概念。通过设置不同的通道，你可以将不同的 App 快照对应到不同的设备。默认，有三个通道：生产、预发布和开发。你可以根据需要添加更多的通道（见图 12-2）。

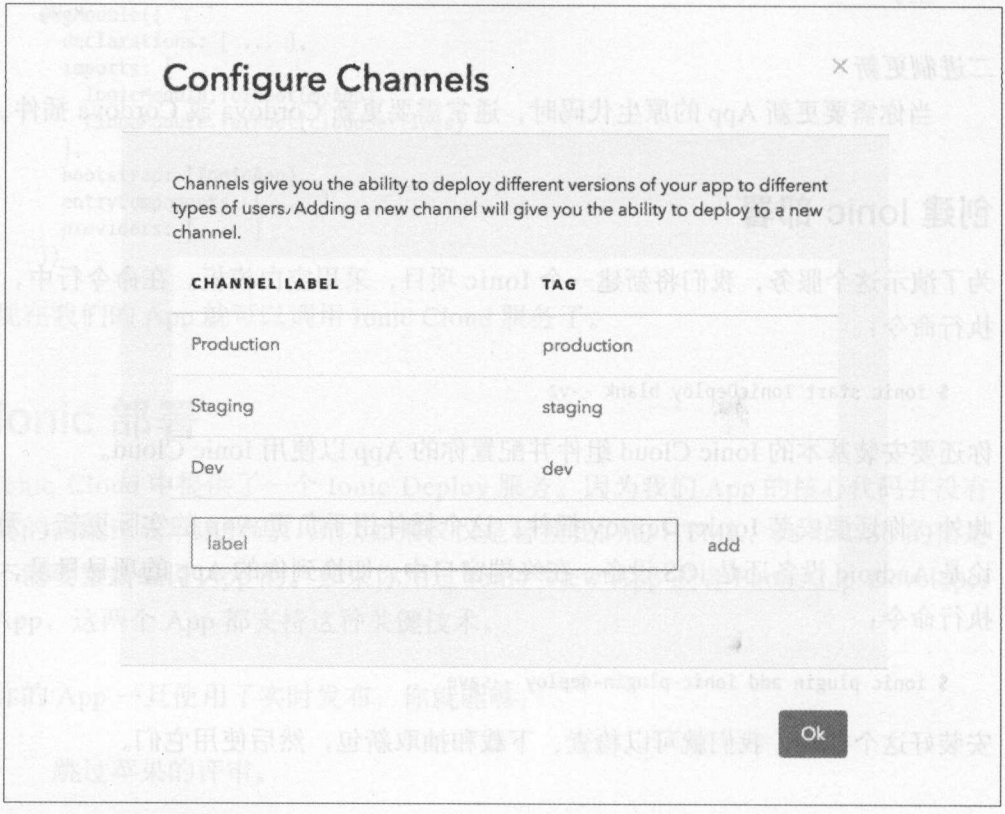



图 12-2：配置通道对话框

在我们的构造器函数中，让我们将部署通道设置为开发。

```
this.deploy.channel = 'dev';
```

这会告诉 Ionic Deploy 在查找快照时只使用这个通道。如果不指定通道的话，默认使用生产。

这个 App 要实现实时发布，还需要进行以下步骤：

1. 检查是否有新快照发布。

2. 应用新快照。

3. 重新加载 App。

### 检查是否有新快照发布

要从 Ionic Cloud 中检查是否有新快照，我们只要简单调用 `deploy.check()`。因为这是一个网络调用，我们应当把它放在一个承诺中进行：

```
this.deploy.check().then((snapshotAvailable: boolean) => {
  // 当 snapshotAvailable 为 true 时，你可以应用新快照
  if (snapshotAvailable) {
    alert('Update is Available');
  } else {
    alert("No Updates Available");
  }
}).catch ((ex) => {
  console.error('Deploy Check Error', ex);
});
```

当 `check()` 方法被调用时，它会返回一个布尔值表示是否需要下载新快照。注意在整个过程中我们会用 alert 来进行演示。

如果存在有效的快照，我们可以调用 `download()` 方法进行下载。这个调用同样要包装在承诺中：

```
this.deploy.download().then(() => {
  alert('Downloading new snapshot');
}).catch((ex) => {
  console.error('Deploy Download Error', ex);
});
```

当快照下载到设备后，我们可以抽取快照：

```
this.deploy.extract().then(() => {
   alert('Extracting snapshot');
}).catch((ex) => {
  console.error('Deploy Extract Error', ex);
});
```

在 extract() 方法的承诺被兑现时，你可以立即调用 load() 方法重启应用。而且，新快照将在 App 下次运行时执行。出于演示的目的，我们会立即调用自动重启，当然我们还是强烈建议给用户一个机会来控制是否重启：

```
this.deploy.extract().then(() => {
   alert('Extracting snapshot');
   this.deploy.load();
}).catch((ex) => {
  console.error('Deploy Extract Error', ex);
});
```

完整的代码如下：

```
this.deploy.check().then((snapshotAvailable: boolean) => {
  // 当 snapshotAvailable 为 true 时，你可以应用新快照
  if (snapshotAvailable) {
    alert('Update is Available');
    this.deploy.download().then(() => {
      alert('Downloading new snapshot');
    this.deploy.extract().then(() => {
      alert('Extracting snapshot');
      this.deploy.load();
      }).catch((ex) => {
        console.error('Deploy Extract Error', ex);
      });
      // this.deploy.extract(); 结束
    }).catch((ex) => {
      console.error('Deploy Download Error', ex);
    });
  } else {
    alert("No Updates Available");
  }
}).catch ((ex) => {
  console.error('Deploy Check Error', ex);
});
```

写好我们的实时发布代码之后，保存文件，接下来让我们创建我们的第一个快照。

### 创建快照

在创建快照的时候，*www* 目录中的文件会被上传到 Ionic 服务器。也就是说，在我们执行 `$ ionic upload` 之前，我们需要编译这个目录。最简单的做法就是执行一下 `$ ionic serve` 命令。并不是想让我们的检查代码在浏览器中运行，我们只是想要编译一下。

要创建快照，我们要用 CLI 将 App 代码上传到 Ionic Cloud。这个命令基本语法如下：

```
$ ionic upload
```

当然，你可以为快照添加一个简单的描述，比如快照想要发布的通道。用以下命令来创建我们的第一个快照：

```
$ ionic upload --note "Initial Deploy" --deploy dev
```

现在，如果你在模拟器中运行 App，你会发现我们的 App 根本不会走完整个更新过程。因为没有下载过任何快照，Ionic Deploy 会检查本地记录，然后会通知系统有一个更新。然后继续往下走，当第一个快照被下载、抽取、应用之后，App 会重启。当我们的代码在检查快照版本时，发现版本匹配直接就退出我们的更新逻辑了。现在让我们小小地修改一下 App，以便我们能够看到真正的更新过程。

打开 *home.html*，修改页面标题为 Ionic Deploy Test。第一行文本来自于“*The Merry Wives of Windsor*”，让我们也将它删除。

保存文件，用 `$ ionic serve` 运行你的 App。然后创建一个新的快照：

```
$ ionic upload --note "Second Deploy" --deploy dev
```

如果你现在再次用 ionic emulate 模拟器上运行或者编译 App，它会重新编译出一个新版本，这样你就看不到实时更新效果了。你需要在另一个模拟器或者在真机上直接点开这个 App。我们的 alert 会再次显示，然后就会看到新的内容（见图 12-3）。

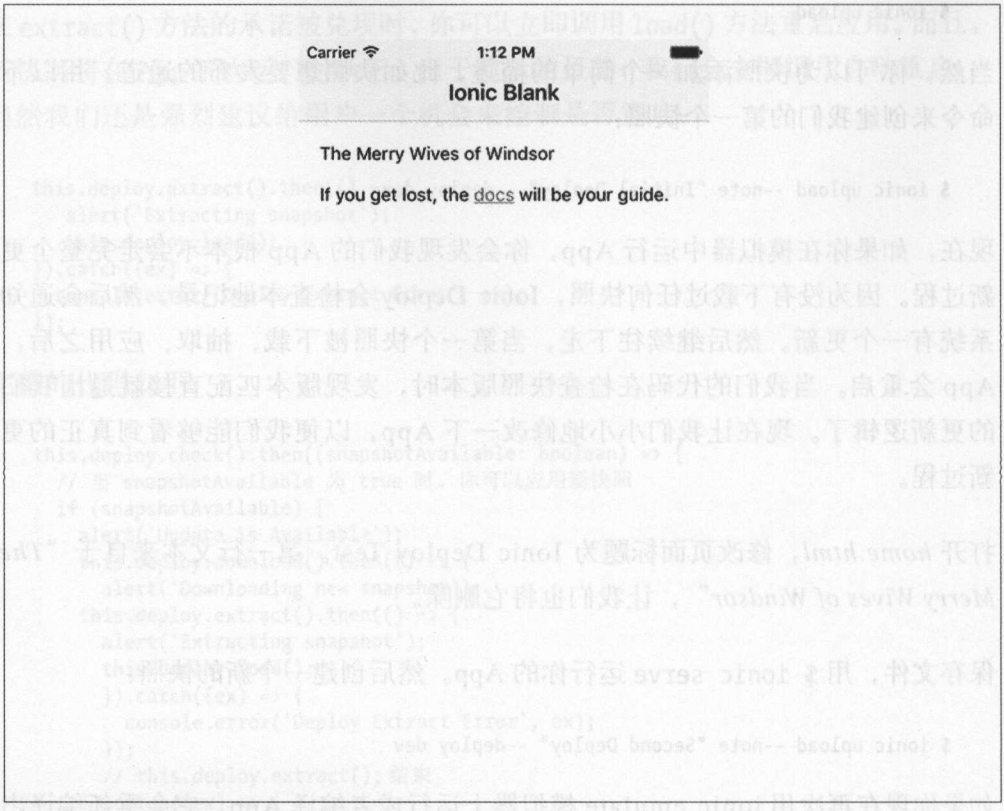



图 12-3：用 Ionic Deploy 更新我们的 App

如果我们打开 Ionic Cloud dashboard，就会看到我们的快照（见图 12-4）。

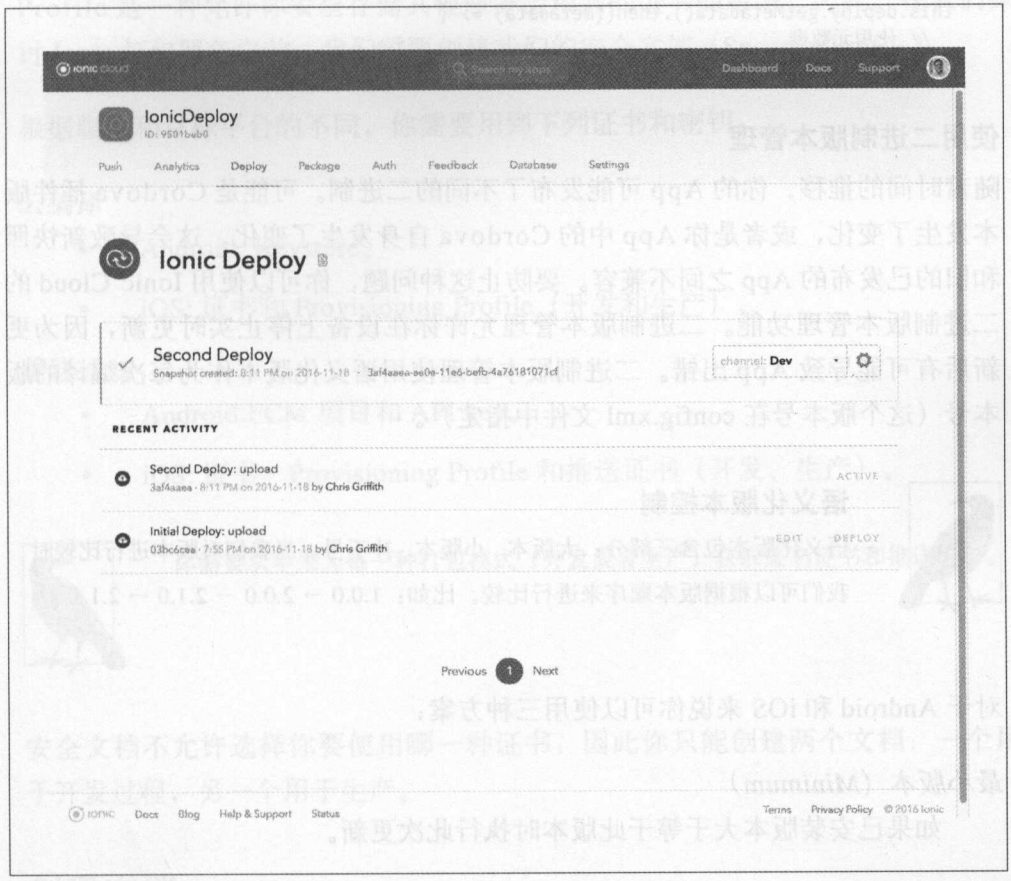



图 12-4：Ionic Deploy dashboard

在这个页面，我们可以管理我们位于不同通道中的部署。如果我们想设置某个快照的通道，也可以在这里进行。在写到这里的时候，还无法从 Ionic Cloud 中删除快照。适当地给你的快照一个描述有助于我们识别不同的快照。你可以用一个示例 App 而不是生产 App 来测试 App 的实时发布。

### 设置快照的元数据

可以在快照中添加自定义的元数据。元数据是通过 dashboard 添加的键值对。例如，你可以在元数据中放入更新说明。

要从 Ionic Cloud 中检索出这些信息，需要调用 `getMetadata()` 方法：

```
this.deploy.getMetadata().then((metadata) => {
  // 使用元数据
});
```

### 使用二进制版本管理

随着时间的推移，你的 App 可能发布了不同的二进制。可能是 Cordova 插件版本发生了变化，或者是你 App 中的 Cordova 自身发生了变化。这会导致新快照和旧的已发布的 App 之间不兼容。要防止这种问题，你可以使用 Ionic Cloud 的二进制版本管理功能。二进制版本管理允许你在设备上停止实时更新，因为更新后有可能导致 App 出错。二进制版本管理使用语义化版本作为每次编译的版本号（这个版本号在 config.xml 文件中指定）。

#### 语义化版本控制

语义化版本包含三部分：大版本 . 小版本 . 补丁号。当我们对版本进行比较时，我们可以根据版本顺序来进行比较。比如：1.0.0 → 2.0.0 → 2.1.0 → 2.1.1。

对于 Android 和 iOS 来说你可以使用三种方案：

*最小版本（Minimum）*

如果已安装版本大于等于此版本时执行此次更新。

*最大版本（Maximum）*

如果已安装版本小于等于此版本时执行此次更新。

*等于（Equivalent）*

如果已安装版本等于此版本，不执行此次更新。注意在发布到 App 商店时请设置为此值，这时候不需要实时更新。

经过一小点准备之后，你就在你的 App 中引入了一个强大的、快捷的实时更新方案。

## 安全文档

在使用 Ionic 打包服务时，Ionic Cloud 会用到各种签名证书和密钥。Security

Profile 是一种允许你安全存储其他服务会用到的证书的服务。因此，在我们探讨 Ionic 打包服务之前，我们需要创建我们的安全文档（Security Profile）。

根据服务和目标平台的不同，你需要用到下列证书和密钥：

*云编译*

- Android: Keystore。
- iOS: 证书和 Provisioning Profile（开发和生产）。

*推送通知*

- Android:FCM 项目和 API key。
- iOS: 证书、Provisioning Profile 和推送证书（开发、生产）。

你需要从苹果为每一种打包模式（开发或者生产）获取签名证书和推送证书。

安全文档不允许选择你要使用哪一种证书，因此你只能创建两个文档：一个用于开发过程，另一个用于生产。

## 创建文档

登录 Ionic.io 官网，选择要创建的文档的 App。因为签名证书和密钥是和特定 App ID 进行绑定的，因此安全文档也和 App 绑定。选择一个 App 之后，进入 Settings 面板并选择 Certificates，如图 12-5 所示。

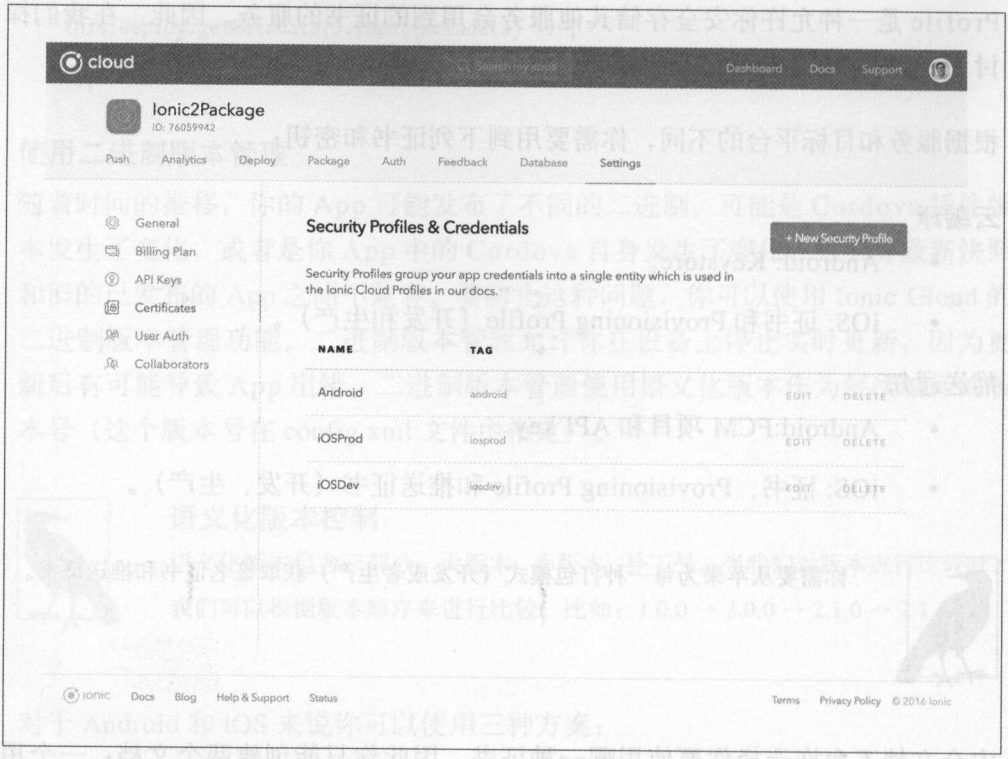



图 12-5：安全文档和证书 dashboard

单击“New Security Profile”按钮，打开新建安全文档对话框。设置文档名称。Ionic Cloud 会根据文档名称来生成一个标签。这个标签会在命令行中用到，因此空格会被下划线替换，所有字母都会转换成小写。安全文档是通过标签来引用而不是文档名来引用的。注意你这个名字不允许在以后更改。因此随意输入一个和描述性文字稍有不同的名字。一个安全文档既支持 iOS 也支持 Android（见图 12-6）。

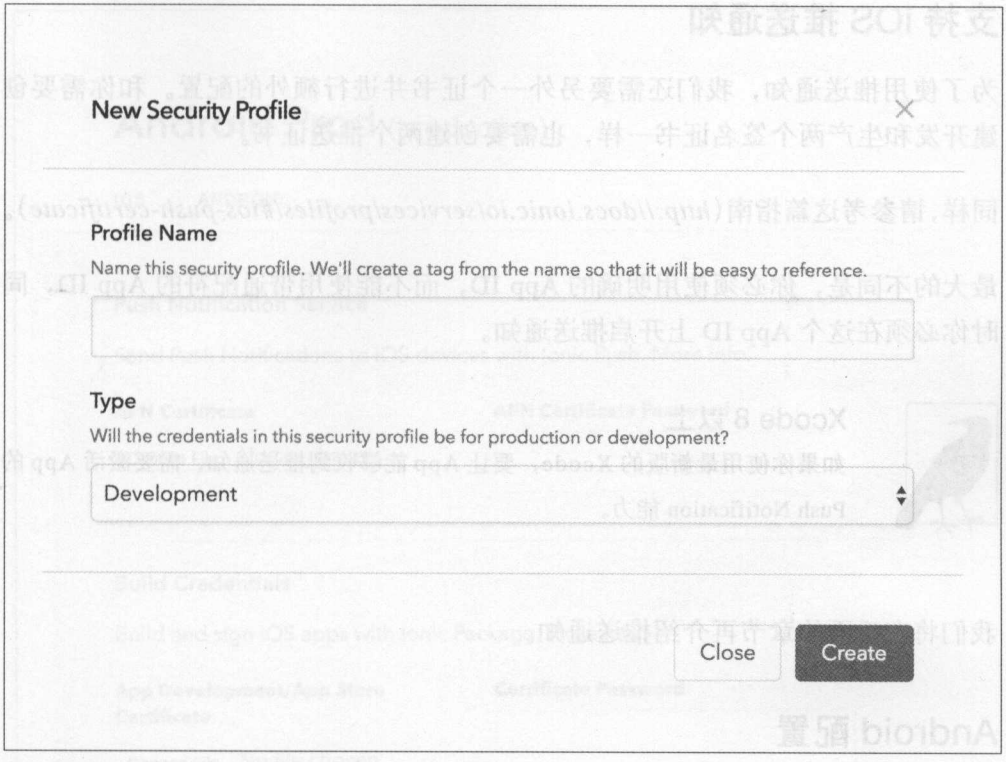



图 12-6：新建安全文档对话框

## iOS 配置

你需要创建 App 的开发 / 商店证书、激活文档（provisioning Profile），并将它们上传到安全文档。这些文件通过苹果开发者中心网站、钥匙串工具（macOS 系统）或者 OpenSSL（Windows 系统）来生成。如果不想重复这些步骤，你可以参考这篇指南（*http://docs.ionic.io/services/profiles/#ios-setup*）来生成 *.p12* 和 *.mobileprovision* 文件。

如果你想创建一个 iOS App 发布证书，并发现这个选项不可用，那是因为每个团队只能拥有一个有效的发布证书。找到已有的证书，导出它，然后继续。

下载完这些文件，我们就可以将它们上传到我们的安全文档中了。

## 支持 iOS 推送通知

为了使用推送通知，我们还需要另外一个证书并进行额外的配置。和你需要创建开发和生产两个签名证书一样，也需要创建两个推送证书。

同样，请参考这篇指南（*http://docs.ionic.io/services/profiles/#ios-push-certificate*）。

最大的不同是，你必须使用明确的 App ID，而不能使用带通配符的 App ID，同时你必须在这个 App ID 上开启推送通知。

**Xcode 8 以上**

如果你使用最新版的 Xcode，要让 App 能够收到推送通知，需要激活 App 的 Push Notification 能力。

我们将在后面的章节再介绍推送通知。

## Android 配置

Android 不需要从 Web 网站上创建签名密钥。取而代之，它使用的是 Java SDK 中的 keytool 工具。

```
$ keytool -genkey -v -keystore MY-RELEASE-KEY.keystore -alias MY_ALIAS_NAME
-keyalg RSA -keysize 2048 -validity 10000
```

将 *MY-RELEASE-KEY* 和 *MY_ALIAS_NAME* 换成 App 的对应值。这个工具会询问你一个 keystore 密码和一个密钥密码。请记住生成这个 keyStore 时你所提供的别名（alias）。

然后，直接将 *.keystore* 文件上传到你的安全文档（见图 12-7）。

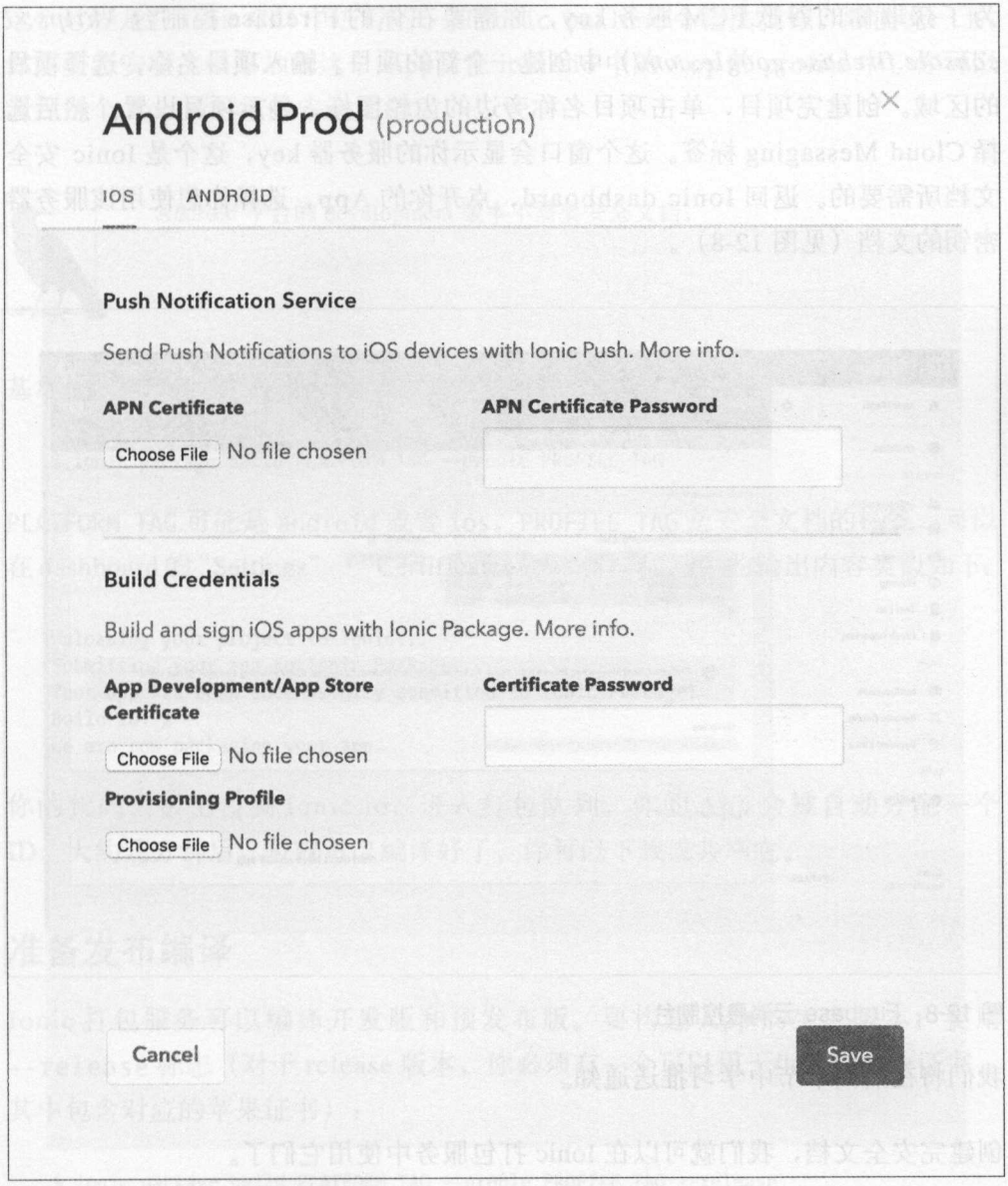



图 12-7：Android 的安全文档对话框

## Android 的推送通知

和 iOS 一样，为了使用推送通知，还需要一些额外的配置。Google 最近将它的云消息服务从 Google 云消息（Google Cloud Messaging，GCM）重新命名为 Firebase 云消息（Firebase Cloud Messaging，FCM）。

为了获取你的谷歌 FCM 服务 key，你需要在你的 Firebase 控制台（*https://console.firebase.google.com/*）中创建一个新的项目。输入项目名称，选择项目的区域。创建完项目，单击项目名称旁边的齿轮图标，显示项目设置。然后选择 Cloud Messaging 标签。这个窗口会显示你的服务器 key，这个是 Ionic 安全文档所需要的。返回 Ionic dashboard，点开你的 App，选择你想使用该服务器密钥的文档（见图 12-8）。

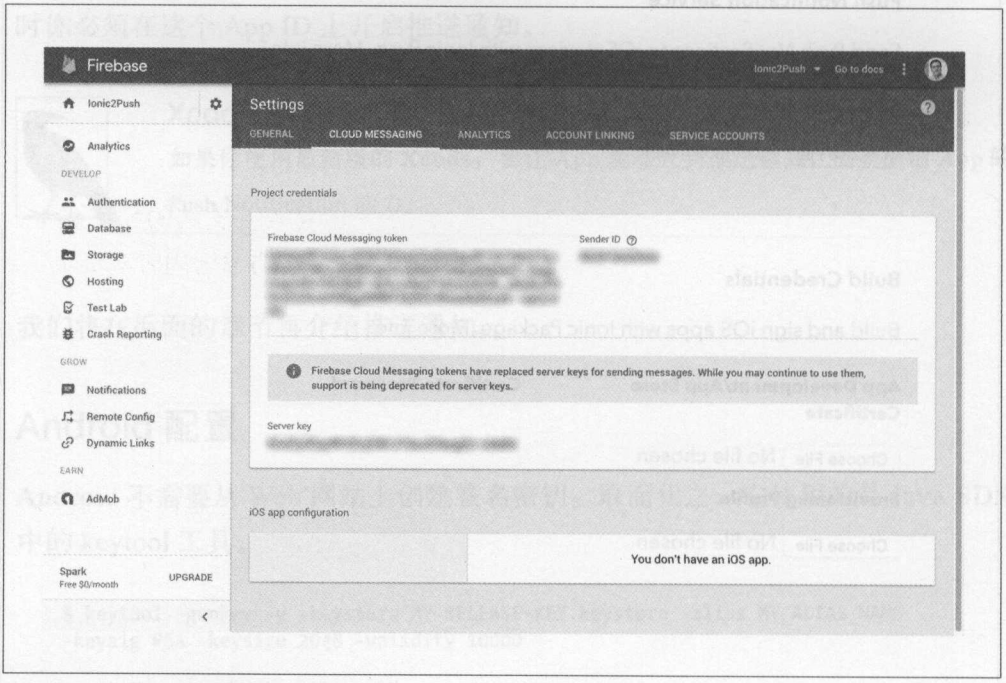



图 12-8：Firebase 云消息控制台

我们将在后面章节中学习推送通知。

创建完安全文档，我们就可以在 Ionic 打包服务中使用它们了。

## Ionic 打包

Ionic 打包允许你在不需要安装原生 SDK 的情况下编译 App 的原生二进制。这对于想编译 iOS 平台的 Windows 平台的开发者来说太有用了。这个服务和 Adobe 的 PhoneGap Build 非常像。

这个过程是在 Ionic CLI 中进行的，和 `ionic build` 命令语法类似。顺便说一下，你需要将安全文档添加到这个 App。在 ionic.io 中的 App 的 profile 中，你可以看到这个 App 所对应的安全文档。

Android 平台的 development 版本不需要安全文档。

基本的打包命令如下：

```
$ ionic package build PLATFORM_TAG --profile PROFILE_TAG
```

`PLATFORM_TAG` 可能是 `android` 或者 `ios`。`PROFILE_TAG` 是安全文档的标签，可以在 dashboard 的“Settings”→“Certificates”中查看。终端中的输出内容类似如下：

```
Uploading your project to Ionic...
Submitting your app to Ionic Package...
Your app has been successfully submitted to Ionic Package!
Build ID: 1
We are now packaging your app.
```

你的代码会被上传到 Ionic.io，进入打包队列。你的 App 会被自动分配一个 ID，大约几分钟后，包就可以编译好了，你可以下载或共享它。

## 准备发布编译

Ionic 打包服务可以编译开发版和预发布版。要构建 App 的生产版本，要用 `--release` 标志（对于 release 版本，你必须有一个可以用于生产的安全证书，其中包含对应的苹果证书）：

```
$ ionic package build PLATFORM_TAG --profile PROFILE_TAG --release
```

## 读取编译信息

当你的打包请求正在排队中的时候，可能要等上几分钟才会完成，你可以通过 list 的方式来检查它们的状态：

```
$ ionic package list
```

将这个操作放到一个进行中的打包过程中时，会更有用些。此时的输出结果会是这个样子：

```
 id | status   | platform | mode
----+----------+----------+--------
  1 | BUILDING | android  | debug

Showing 1 of your latest builds.
```

## 获取编译结果

当编译完成，你可以用这个命令获得它的状态：

```
$ ionic package info BUILD_ID
```

一个成功的编译信息会是这个样子：

```
id          | 1

status      | SUCCESS

platform    | android

mode        | debug

started     | Nov 19th, 2016 14:43:08
completed   | Nov 19th, 2016 14:43:29
```

## 下载编译文件

当 App 打包完成，就可以下载 *.ipa* 和 *.apk* 文件了。这些文件会下载到 App 的项目文件夹中：

```
$ ionic package download BUILD_ID

Downloading... [============================]  100%  0.0s
Wrote: /Users/chrisgriffith/Desktop/Ionic Book Apps/
       Ionic2Package/Ionic2Package.apk
Done!
```

你现在可以在具有对应权限的设备上安装你的 App 了。

## 更新你的 Cordova 插件

要确保 Ionic Package 的编译服务器知道你的 App 会使用哪些 Cordova 插件，你需要用 --save 标志重新将它们安装到项目中去：

```
$ ionic plugin add PLUGIN_NAME --save
```

要知道 App 中所用到的插件列表，可以运行：

```
$ ionic plugin ls
```

# Ionic View

Ionic View 使你的 App 更容易分享给你的用户和测试人员，既不需要使用 TestFlight 的 beta 测试也不需要使用 Google Play 的 beta 测试。只需要把你的 App 上传并分享。这个免费的移动 App 支持 iOS 和 Android。它充当了一个能够运行你的 App 的原生外壳。这种方法的好处是，你不需要为了在真机上运行 App 而进行编译。

Ionic View 使用了 Ionic Deploy 特性来实现这一点。因为它将你的 App 放在另一个外壳中，因此只能支持对 Cordova 插件中的部分子集进行调用。

## 支持的插件

这是 Ionic View 目前支持的插件列表：

- ActionSheet (2.2.2)
- BarcodeScanner (5.0.0)
- BLE (1.1.1)
- Bluetooth Serial (0.4.5)
- Calendar (4.5.0)
- Camera (2.2.0)
- Capture (1.3.0)
- Contacts (2.1.0)

- DatePicker (0.9.3)
- DBMeter (1.0.3)
- Device (1.1.2)
- Device Motion (1.2.1)
- Device Orientation (1.0.3)
- Dialogs (1.2.1)
- Email Composer (0.8.3)
- Geolocation (2.2.0)
- Globalization (1.0.3)
- ImagePicker (1.1.1)
- Keyboard (2.2.0)
- Media (2.3.0)
- Network Information (1.2.1)
- SocialSharing (5.1.1)
- SQLite (1.4.2)
- StatusBar (2.1.3)
- Toast (2.5.2)
- Touch ID (3.2.0)
- Vibration (2.1.1)

关于你的 App 能够使用的插件的最新列表，请浏览 Ionic 官网（*https://docs.ionic.io/tools/view/#supported-plugins*）。

## 上传你的 App

要在 Ionic View 中运行你的 App，你必须将它上传到 Ionic.io，命令如下：

```
$ ionic upload
```

你的 App 会上传到 Ionic.io，终端窗口会显示上传状态：

```
Uploading app....
Saved app_id, writing to ionic.io.bundle.min.js...
Successfully uploaded (76059942)

Share your beautiful app with someone:
$ ionic share EMAIL

Saved api_key, writing to ionic.io.bundle.min.js...
```

## 查看你的 App

现在直接点开你的 Ionic View App，它会自动列出你已经上传的 App 的列表。你也可以手动输入分享给你的那个 App ID。

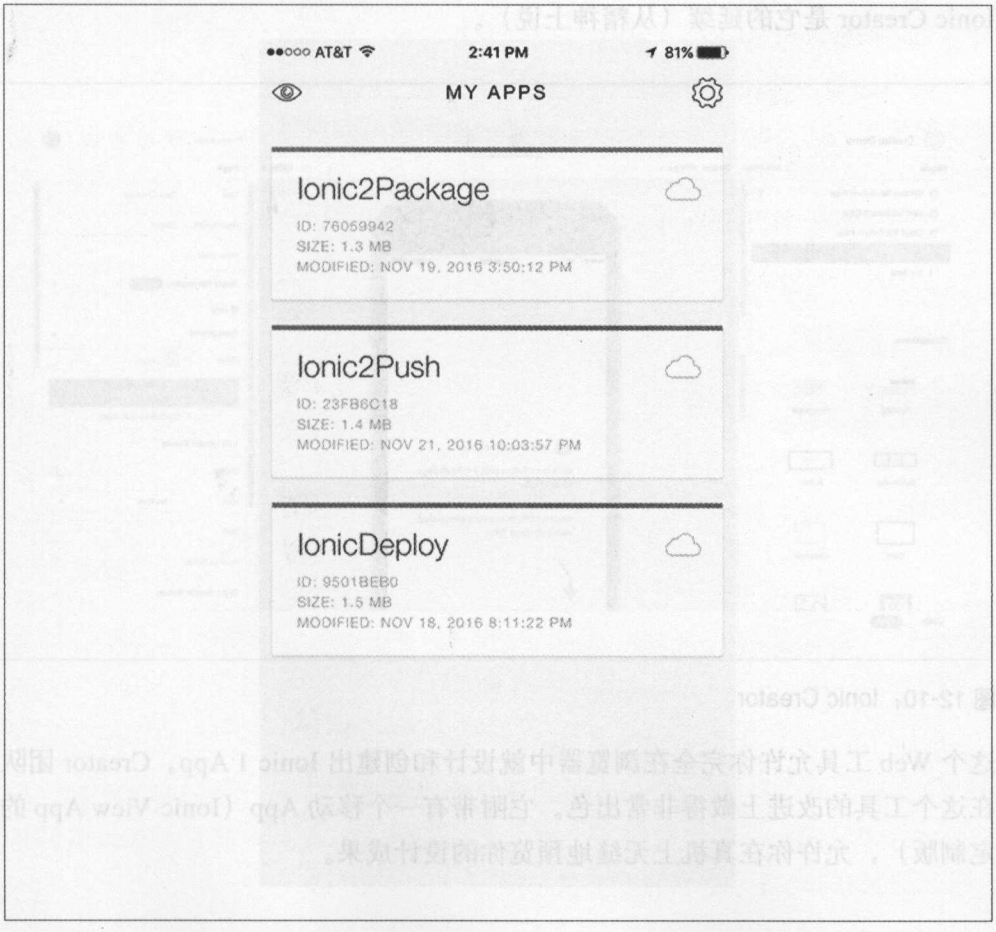



图 12-9：Ionic View 中有效的 App

单击某张 App 卡片，会弹出一个菜单列出可以进行的操作。你可以查看 App、清除它的本地文件、将本地文件和 Ionic Cloud 最新上传的版本进行同步，或者将它从你的账号上删除。

Ionic View 是一个在真机上快速测试和预览你的 App 的强大的工具，并且不需要对 App 进行编译或传统的安装。

## Ionic Creator

Ionic 还有一个东西需要简单提一下。在浏览 Ionic 网站时，你可能看到一个词叫做 Ionic Creator（见图 12-10）。在 Drifty（Ionic 最终的开发团队）开发 Ionic 框架时，他们针对 jQuery Mobile 开发了一个可视化设计工具 Codiqa。Ionic Creator 是它的延续（从精神上说）。

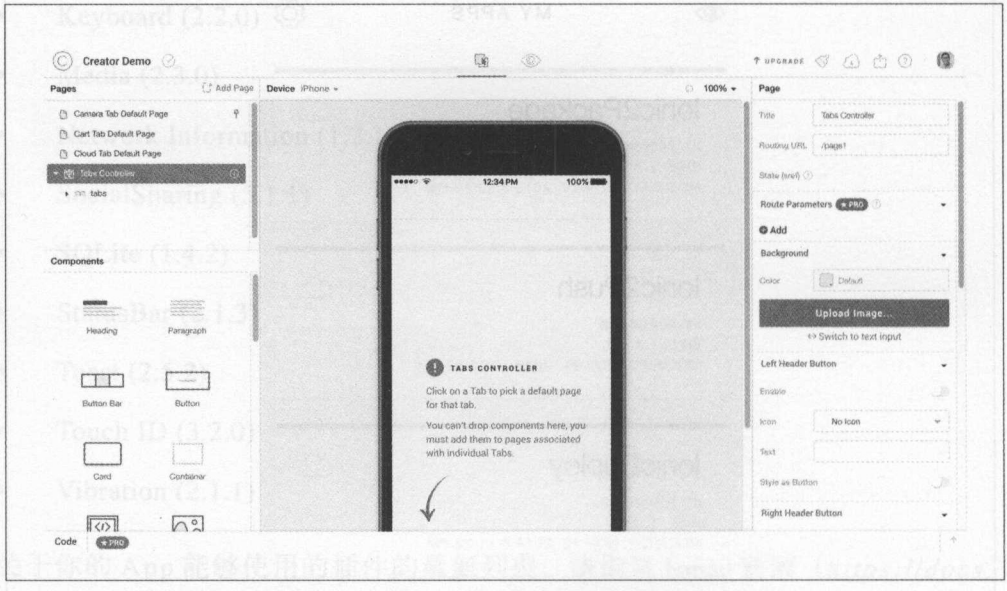



图 12-10：Ionic Creator

这个 Web 工具允许你完全在浏览器中就设计和创建出 Ionic 1 App。Creator 团队在这个工具的改进上做得非常出色。它附带有一个移动 App（Ionic View App 的定制版），允许你在真机上无缝地预览你的设计成果。

### Ionic Creator 只支持 Ionic 1

不幸的是，Ionic Creator 目前只支持 Ionic 1。它的源文件和示例代码在 Ionic 2 项目中是不可用的。开发团队还没有公开什么时候发布兼容 Ionic 2 的版本。

这个工具对于喜欢可视化设计的人来说，是一个学习 Ionic 1 框架的好途径。哪怕你在用 Ionic 2 开发，你仍然可以用这个设计工具来创建预览版和原型。

## 小结

Ionic 服务在核心框架之外提供了许多额外的功能。尽管它们不是免费的，它们和框架的结合是如此的紧密，你可以根据实际开发需求做出适当的选择。

# 第 13 章

# 渐进式 Web 应用

因为 Ionic App 用 Web 技术构建，你可能会好奇，它能否以传统的 Web App 的方式进行部署，并使用标准浏览器进行浏览呢？就像在编写和测试我们的 App 时我们所用的 `$ ionic serve` 命令一样。你当然可以将 *www* 目录中的文件放到你的 Web 服务器上。这样做的一个最大问题是在这种环境下 Cordova 插件无法工作。用这种方式部署 App 的需要好好考虑一个全新的平台：浏览器。

虽然我们不具备和原生 App 一样的能力，但当下的现代浏览器都会支持几个重要特性，比如定位、通知和离线存储。

要将你的 App 以这种方式分发有一个难题，就是我们的 Ionic App 的文件的整体大小。*main.js* 文件是用 CLI（`$ ionic build --prod`）生成的，一开始就超过了 2 MB。

**减少未来 build 的大小**

Ionic 团队正在设法让 Angular 减少 App 文件大小。

你接下来肯定就会想到那些 App 中用到的图形和其他资源，你的 App 可能运行起来非常慢，因为这些资源每一个都需要从网络上下载。尤其是对于蜂窝网络或糟糕的 WiFi 网络，这个问题更显严重。

但是，在 Web App 开发社区出现了一个新趋势，即渐进式 Web App。这个概念最初由 Google 在 2015 年提出，这种方案充分利用了新技术的特点，将 Web App 和移动 App 二者的优点结合在一起。当然，不是每种浏览器都能完全支持渐进式 Web App 的所有功能，但可以将它作为一种备选的技术方案。

## 但是，什么是渐进式 Web App？

根据 Google 开发者网站（*http://bit.ly/2mKBYZv*）的描述，渐进式 Web App 利用现代 Web 技术产生接近 App 的用户体验。

首先来解释第一个词，渐进式（progressive）。根据定义，一个渐进式 Web App 必须能够在任意设备上运行，同时可以用渐进的方式增强它的功能。比如，如果设备有足够的能力，App 就能调用这些能力。如果某个特性不支持，Web App 会提供一种替代的体验。

渐进式 Web App 同时是可被发现和可连接的，也就是说，用户和搜索引擎都能够搜索出 App 中的内容。很显然原生 App 是没有可被发现的特点的。因为我们的 App 存在于 Web 上，用户可以跳转到或者收藏他们在 App 中的“位置”，然后可以回到这些位置。

因为我们正在不断突破现有屏幕尺寸的范围，这些 App 必须对视觉设计进行适应。实际上，就算手机屏幕大小固定，也还有许多设备的屏幕尺寸迫使我们必须正视这个问题。

就视觉设计来说，一个渐进式 Web App 从外观上和原生 App 没有区别。不得不说，Ionic 的组件让我们事半功倍。

我们还需要脱离浏览器，将它变成某种用户可以安装到设备的主屏上的东西。否则的话用户会觉得使用你的 App 太麻烦了，因为他们需要点开浏览器，输入 URL 或者点开某个书签。

考虑到安全的因素，一个渐进式 Web App 必须使用 HTTPs 连接才能工作。

最后，它必须不用网络也能工作。当原生 App 安装好后，它就能使用。渐进式 App 必须提供一样的能力。

现在，你已经了解了渐进式 Web App，接下来我们看一下如何将我们 Ionic App 转变成一个渐进式 Web App。

## manifest.json 文件

我们曾经对 Ionic App 的文件和目录进行过一番探究，有两个文件我们直接跳过了：*manifest.json* 和 *service-worker.js* 文件。

首先，让我们的 Web App 更像原生 App。这样，我们就需要用到 *manifest.json* 文件。这个文件符合 W3C 的 Web App 规范（*https://w3c.github.io/manifest/*）。这是自动生成的默认的 *manifest.json* 文件内容：

```
{
  "name": "Ionic",
  "short_name": "Ionic",
  "start_url": "index.html",
  "display": "standalone",
  "icons": [{
    "src": "assets/imgs/logo.png",
    "sizes": "512x512",
    "type": "image/png"
  }],
  "background_color": "#4e8ef7",
  "theme_color": "#4e8ef7"
}
```

让我们具体介绍一下以下属性：

*name*

用于作为 App 的显示名称。

*short-name*

通常用于和 Home 屏上的 App 图标一同显示。

*start_url*

App 的入口，通常会是 index.html 文件。

*display*

指定浏览器应该如何显示这个 Web App。可选值为：fullscreen、standalone、minimal-ui 或者 browser。每个值有不同作用：

*fullscreen*

这个 Web App 显示时不会显示浏览器外壳，直接占满整个屏幕空间。

*standalone*

这个 Web App 启动后就像一个独立的原生 App。浏览器外壳不显示，但会显示其他系统的 UI 元素比如状态栏或者系统返回按钮。

*minimal-ui*

这个 Web App 会显示一个用于控制导航的 UI 元素集合（比如，返回、前进、刷新，以及某些能够查看 document Web 地址的方法）。

*browser*

这个 Web App 会显示在标准的浏览器外壳中。

*icons*

这个数组用于指定各种图标，用于 Home 屏的、用于任务切换器等。它是一个 object 数组，object 中包含了图标的 url、大小以及文件类型。我们建议提供 48、96、144、192 和 256 像素的图标。

*background_color*

一个十六进制颜色值，用于定义 Web App 的背景色。在 Chrome 中，这个值也同时用作 splash 屏的背景色。

*theme_color*

在 Android 上，这个十六进制颜色值指定了状态栏的颜色。

没有列在这个自动生成的 *manifest.json* 文件中的属性还有 `orientation`。这个属性定义了 App 的默认方向。它的取值可能是 portrait 和 landscape。

这个文件已经链接到我们的 *index.html* 的 head 标签内：

```
<link rel="manifest" href="manifest.json">
```

## Service Workers

每个渐进式 Web App 的内核实际上是 Service Worker API。这个 API 在运行在

App 中的代码和运行在远程服务器上的代码之间形成了一个新的中间层。这些代码在我们的客户端和服务端充当了中间人的角色。当 Service Worker 注册成功，它会独立于浏览器窗口或 tab 标签页而存在。

这个 API 当前支持 Chrome、Firefox 和 Opera，未来还可能包括 Edge。苹果的 WebKit 团队已经将它标记为“正在考虑”。

Service worker 能够拦截和重写你的 App 的网络请求。这样，当数据连接不可用时，你可以提供缓存过的响应数据。因此，它只能在安全上下文中使用（比如 HTTPS）。

在默认的 index.html 中，Ionic 团队已经包含了一小段代码帮你注册 App 的 service worker。默认这段代码是注释掉的：

```
<!-- un-comment this code to enable service worker
<script>
  if ('serviceWorker' in navigator) {
    navigator.serviceWorker.register('service-worker.js')
      .then(() => console.log('service worker installed'))
      .catch(err => console.log('Error', err));
  }
</script>-->
```

这段代码直接注册了 service worker，如果浏览器支持的话。真正的代码是在 *service-worker.js* 中实现的。

自动创建的 *service-worker.js* 的注释非常完美，这里我们就不需要重复介绍了。但是，我们可以参考这个基本的 service worker 的例子，你可以通过它学习如何实现一个 service worker。

让我们将默认的 *servcie-worker.js* 替换掉。在这样做之前，请保存一份 *service-worker.js* 的拷贝。这里是新代码：

```
self.addEventListener('fetch', function(event) {
  if (event.request.url.indexOf('/img/my-logo.png') !== -1) {
    event.respondWith(
      fetch('/img/my-logo-flipped.png')
    );
  }
});
```

这段代码拦截任何对"/img/logo.png"所进行的请求，然后返回这个图片的flipped版本。

记住，为了让这段代码能够运行，它必须使用HTTPS连接，并在兼容的浏览器上运行。

通过缓存资源，我们可以明显减少App的启动时间。事实上，我们可能会比原生App启动还快！

注意，因为service worker是一种特殊的共享Web worker，它会运行独立的线程中，独立于你的主页面的JavaScript运行的线程。也就是说，它会被所有和service worker同一路径的Web页面所共享。例如，一个service worker位于*/my-app/sw.js*，它对*/my-app/index.html*和*my-app/images/ header.jpg*生效，但对*/index.html*无效。

除了拦截网络请求，service worker还提供离线功能、推送通知、后台内容更新、内容缓存等功能。

例如，当你试图在离线状态下访问一个Web页时，你可以提供一个自定义的文字作为响应：

```
self.addEventListener('fetch', function(event) {
  event.respondWith(
    fetch(event.request).catch(function() {
      return new Response(
        'Welcome to the Upside Down.\n'+
        'Please look out for the Demogorgon.\n'+
        'We look forward to telling you about Hawkins National Laboratory
        as soon as you go online.'
      );
    })
  );
});
```

刷新浏览器，当这个service worker被载入后，在离线状态下刷新页面。默认的“无法连接”页面将被这段文字替换。只需要一小点代码，你就可以提供自定义的体验，包括图片和样式。

下面是一个能够缓存大部分默认的Ionic文件并在用户离线时返回缓存过的版本：

```
var CACHE_NAME = 'ionic-cache';
var CACHED_URLS = [
  '/assets/fonts/ionicons.eot',
  '/assets/fonts/ionicons.scss',
  '/assets/fonts/ionicons.svg',
  '/assets/fonts/ionicons.ttf',
  '/assets/fonts/ionicons.woff',
  '/assets/fonts/ionicons.woff2',
  '/build/main.css',
  '/build/main.js',
  '/build/polyfills.js',
  'index.html'
];

self.addEventListener('install', function(event) {
  event.waitUntil(
    caches.open(CACHE_NAME).then(function(cache) {
      return cache.addAll(CACHED_URLS);
    })
  );
});

self.addEventListener('fetch', function(event) {
  event.respondWith(
    fetch(event.request).catch(function() {
      return caches.match(event.request).then(function(response) {
        return response;
      });
    })
  );
});
```

Service woker 极大地改善了 App 的用户体验。除了提供一种允许我们的 App 工作在离线模式下的办法，它还能用于用户网络连接较差的时候。

试想一下，你其实有许多 App 实际上是非常静态的，你可以始终使用本地而不是网络请求来提供服务。

# 推送通知

用于桥接在原生 App 和 Web 之间的一个新的 API 就是推送通知。这个 API 为了在浏览器没有打开的情况下接收通知，也使用了 service worker。关于在你的渐进式 Web App 中添加推送通知的具体内容，请参考 Google 教程（*http://bit.ly/2mKVvZX*）。

## 小结

对渐进式 Web App 的支持随着 Ionic 框架的发展而持续增长，请随时了解新的改变。Ionic 有一个渐进式 Web App 示例的列表（*https://pwa.ionic.io/*）。

要了解更多关于渐进式 Web App 的内容，建议阅读由 Tal Ater 编写的《Building Progressive Web Apps: Bringing the Power of Native to the Browser》（O'Reilly 出版）一书。

# 第 14 章

# 终章

在最后一章，我们会介绍几个在示例程序中没有用到的组件，并探讨在你的 Ionic 框架学习之旅中，后续还应该学习什么。

尽管我们已经接触过一些 Ionic 库的组件，但我们仍然有一些其他的组件需要重点介绍。更完整的组件列表，请看附录 C。

## Slides

第一个要介绍的组件是 Slides。这个组件允许你以轮播的形式显示一系列内容。要显示的内容可以是简单图片，也可以是几张复杂的页面。这个组件实际上由两部分构成：容器组件 `<ion-slides>` 及其所包含的单个的 `<ion-slide>`。在 Ionic 文档中有一个很简单的例子：

```
<ion-slides pager>

  <ion-slide style="background-color: green">
    <h2>Slide 1</h2>
  </ion-slide>

  <ion-slide style="background-color: blue">
    <h2>Slide 2</h2>
  </ion-slide>

  <ion-slide style="background-color: red">
    <h2>Slide 3</h2>
  </ion-slide>

</ion-slides>
```

出于演示的目的，这里使用的是行内 CSS 样式。

表 14-1 列出了 `<ion-slides>` 支持的各种属性。

表 14-1：<ion-slides> 的设置

| 属性 | 类型 | 默认值 | 描述 |
| --- | --- | --- | --- |
| autoplay | 数字 | - | 两次切换之间的延迟（单位：毫秒）。如果不指定这个参数，自动播放禁止 |
| direction | 字符串 | 'horizontal' | 滑动方向：'horizontal' 或者 'vertical' |
| initialSlide | 数字 | 0 | 第一张幻灯片的索引 |
| loop | 布尔值 | false | 是否无限循环，当播放到最后一张后又从第一张开始 |
| pager | 布尔值 | false | 显示小圆点 |
| speed | 数字 | 300 | 两张幻灯片之间的切换动画的时长（单位：毫秒） |

这个组件实际上是一个对来自于 iDangero.us 的 Swiper 组件的封装。默认，这个组件直接暴露出来的一些方法被 Ionic 团队限制了。如果你需要访问 iDangero.us 原来的功能，你需要用 `getSlider()` 方法获的一个 Swiper 实例，然后调用这个实例的方法。

如果你想在示例 App 中使用这个组件，你可以将每个城市的天气预报放在里面，或者用于显示每个国家公园的图片。

## Date-Time

使用日期和时间不仅仅是编码的事情（这部分工作你可以用 *moment.js*），还和用户界面有关。尽管 HTML5 有一个特殊的 input 类型 dateime-local，但这个组件的可视化界面是不确定的，而且很难定制化。因此 Ionic 用 `<ion-datetime>` 来解决这个问题：

```
<ion-datetime displayFormat="h:mm A" pickerFormat="h mm A"
[(ngModel)]="event.timeStarts"></ion-datetime>
```

你可以控制它的外观和日期时间的选择器格式（在 *http://bit.ly/2mKWMAo* 上有很多选项）。这些格式通过标准过滤器来定义。在上面的代码中，小时将以

1~12 来显示，然后是冒号，然后是两位数的分钟，不足两位补 0。这个日期选择器的小时的取值范围是 1~12，分钟为 00~59，最后是 AM 或 PM（见图 14-1）。

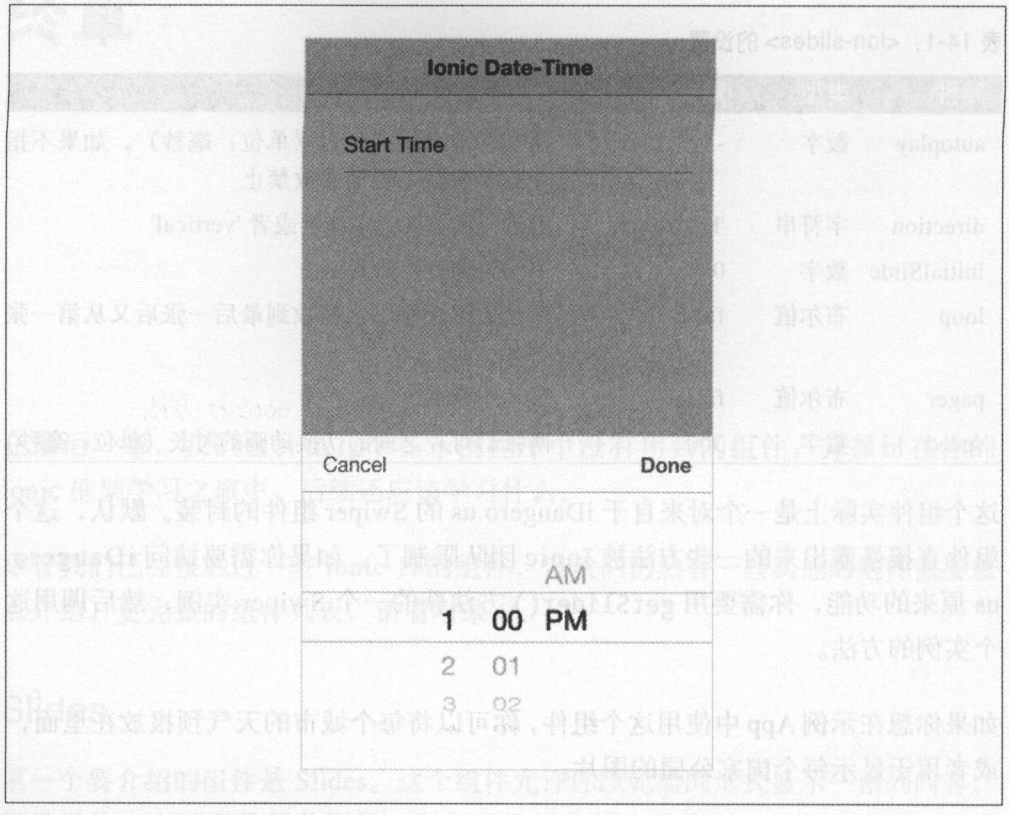



图 14-1：Ionic 的 Date-Time 组件

你可以在第 7 章的 ToDo App 中用这个组件对待办事项的过期时间进行设置。

## Popover

你经常需要向用户显示某些对象的附加信息，或者它的进一步的控制，但又不想破坏 App 的界面。这时 Popover 组件就派上用场了。这个组件的常见例子是设置列表的排序方式或过滤条件。和 HTML 模板的组件不同，这个组件是动态创建的，和 Ionic Alert 组件非常像：

```
import { Component } from '@angular/core';
import { PopoverController } from 'ionic-angular';
import { MyPopOverPage } from './my-pop-over';
```

```
@Component({
  selector: 'page-home',
  templateUrl: 'home.html'
})

export class HomePage {

  constructor(public popoverCtrl: PopoverController) {

  }

  presentPopover(ev) {
    let popover = this.popoverCtrl.create(MyPopOverPage);
    popover.present({
      ev: ev
    });
  }
}
```

在我们的 HTML 中，我们用一个简单按钮调用 presentPopover 方法：

```
<button ion-button (click)="presentPopover($event)">
  <ion-icon name="more"></ion-icon>
</button>
```

真正的 popover 组件的实现是在这里（见图 14-2）：

```
import { Component } from '@angular/core';
import { ViewController } from 'ionic-angular';

@Component({
  template: `
    <ion-list>
      <ion-list-header>Units</ion-list-header>
      <button ion-item (click)="close()">Celsius</button>
      <button ion-item (click)="close()">Fahrenheit</button>
    </ion-list>
  `
})
export class MyPopOverPage {

  constructor(public viewCtrl: ViewController) {}

  close() {
    this.viewCtrl.dismiss();
  }
}
```

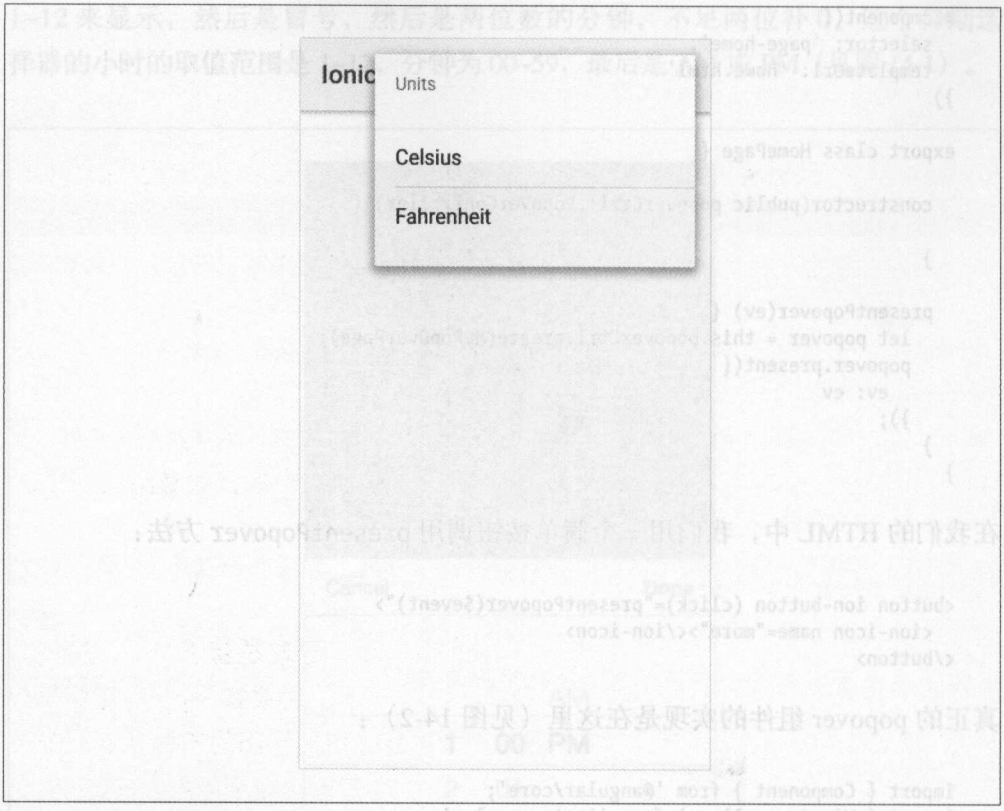



图 14-2：Ionic Popover 组件

最后一件事情是修改你的 *app.module.ts* 文件，添加你的 Popover 组件。如果你忘记这点，你的组件无法工作。

如果你想对列表内容进行排序，以 Ionic2Do App 为例，你可以用这个组件给用户提供一个排序选项。另外，你还可以用这个组件来标记我们的温度单位。

## Reorder List

另一个我们要介绍的组件是 ItemReorder。它和标准 Ionic List 组件结合在一起能允许用户对列表进行重排。要实现这一点，直接在 `<ion-list>` 中添加一个 reorder 属性并设置属性值为 true。这会在列表项中暴露出 reorder 控件。

Ionic 会为你处理数组的重排。如果出于某种目的你需要自己处理重排，你可以将 `ionItemReorder` 事件绑定到自定义函数上：

```
<ion-list reorder="true" (ionItemReorder)="reorderItems($event)">
  <ion-item *ngFor="let item of items">{{ item }}</ion-item>
</ion-list>
```

当事件被触发，它会将被重排的列表项的原始索引（from）和新索引（to）作为参数传递。在下面的 `reorderItems` 函数中，会在控制台中打印出重排后的索引 indexes，进行真正的排序，然后将新的数组打印到控制台：

```
reorderItems(indexes) {
  console.log(indexes);  //{from: 0, to: 4}
  this.items = reorderArray(this.items, indexes);
  console.log(this.items);   // 重排后的数组
}
```

Ionic 有一个工具函数 `reorderArray`，能够省去你保存索引和拼接数组的工作。

## DeepLinker

这是 Ionic 2 中最新的一个组件和特性。在 Ionic 1 中，导航是基于 URL 进行的。虽然在复杂 App 中导致了一些问题，但它仍然提供了一种暴露 App 中更深层次的方法。在 Ionic 2 中使用了新的导航系统，导航到指定界面的不再是那么简单了。对于许多 App 设计者来说这都成为了问题。为了解决这个问题，Ionic 提出了 DeepLinker 系统。

要使用这个特性，我们必须修改 *app.module.ts* 文件。在我们调用 `IonicModule.forRoot` 方法时，第三个参数简单地用一个 links 数组作为参数：

```
imports: [
  IonicModule.forRoot(MyApp, {}, {
    links: [
      { component: HomePage, name: 'Home', segment: 'home' },
      { component: DetailPage, name: 'Detail', segment: 'detail/:userId' }
    ]
  })
]
```

如果你需要向我们要导航到的页面传递参数，我们可以用 `:param` 语法进行。

这个特性允许我们从任意指定页面来打开我们的 App，我们还可以将我们的 App 部署为渐进式 Web App，并添加书签。

## Storage

我们来最后看一个特性，Ionic 内置的存储系统。我们曾经了解过 Firebase 的云存储系统，但如果你只需要保存简单数据（比如用户想查看天气预报的城市列表）用 Firebase 就有点大材小用了。

Ionic 的 Storage 允许你毫不费事地保存键值对和 JSON 数据。使用本地数据存储有许多方式，它们都有各有优劣。Ionic Storage 会使用各种存储方式，并根据 App 所运行的平台自动选择正确的方式。

例如，如果你在 Web 上运行，Ionic 存储首先会尝试使用 IndexedDB 来保存数据。如果浏览器不支持 IndexedDB，它会用 WebSQL 来保存。如果还是不行，它会用 localstorage。如果你的 App 是以原生应用方式来部署的，则可以安装 SQLite 插件然后使用 SQLite 来保存数据。

Ionic Storage 的使用十分简单。首先，如果你想在原生 App 中使用 SQLite，你必须安装 SQLite 的 Cordova 插件：

```
ionic plugin add cordova-sqlite-storage --save
```

这个包已经安装到我们的 node 模块中了，因此我们只需要在 *app.modules.ts* 文件的 providers 数组中声明它：

```
import { Storage } from '@ionic/storage';

@NgModule({
  declarations: [
    // ...
  ],
  imports: [
    IonicModule.forRoot(MyApp)
  ],
  bootstrap: [IonicApp],
  entryComponents: [
    // ...
  ],
  providers: [
    Storage
  ]
})
export class AppModule {}
```

然后我们就可以在我们的 App 中使用 Ionic Storage 了：

```
import { Storage } from '@ionic/storage';

export class MyApp {
  constructor(storage: Storage) {

    // 存入 key/value
    storage.set('name', 'Chris');

    // 读取 key/value
    storage.get('name').then((val) => {
      console.log('Your name is', val);
    })
  }
}
```

现在你明白 Ionic Storage 的使用是有多简单了吧，赶紧把 IonicWeather App 中的城市列表保存下来吧！

# 下一步

本书为学习 Ionic 打下了良好基础，但如果你觉得意犹未尽或者框架发生了升级，或者工具链中某一部分不会用怎么办？你可以在哪里寻求帮助呢？

这里有一些资源，是你首先应该去尝试的。在你将问题发送到论坛或者 Slack 之前，首先看看这些问题是否已经有现成的答案了。一些经常性的问题，比如“为什么我下拉刷新之后屏幕上什么都没有改变？”其实已经有人贴出答案了。但是这个问题本身提得就有点问题。比如你使用的 Ionic 版本是什么？你的测试环境是什么？在解决它的时候你做了哪些尝试？

因此对于这个问题，我们必须看看这个问题到底是想问什么？下拉刷新动作是否已经正确地触发了？如果是，那么后面的函数被调用了吗？继续沿着代码的执行顺序，一路检查下去。

最终，这个问题很可能根本和 pull to refresh 组件无关，而是因为数据提供者返回了缓存数据。但如果仅从最初提出的问题来看，你永远猜不出问题之所在。

## Ionic 论坛

寻求帮助的第一个地方就是 Ionic 自己的论坛。这是帮助你解决所遇到的各种问题的好去处。在这个论坛中有不少成员是 Ionic 团队中的成员和技术社区的专家。

## Ionic 全球 Slack 频道

获得帮助的另一个好去处是 Ionic Worldwide Slack channel（*http://ionicworldwide.herokuapp.com/*）。它不像论坛那样正式，因此你可以在需要提某个问题或者获取解决问题的建议时使用它。

## GitHub

Ionic 框架是开源的，如果你遇到一个真正的 Bug，你可以到它的代码库报告所发现的问题。当然如果你能在修复这个问题后，再提交一个 pull request 就更好了。如果你认为这个问题可能是框架自身导致的，你可以去它的 GitHub 库（*https://github.com/driftyco/*）上找找，看这个问题是否有人发现，以及这个问题是否已经被解决，或者有没有人贴出了某个变通办法。

# 小结

这本书我们就介绍到这里。本书试图以合理顺序和一定深度，从各个方面介绍 Ionic 和相关技术。本书中没有足够的空间和时间来介绍 Ionic 框架的每一部分，但它会为你打下一个坚实的基础，让你快速构建出令人惊讶的、流畅的、高性能的混合移动 App。不断尝试新技术，并和我们一起，将 Ionic 变成一个伟大的框架！

附录 A

# 从 Ionic 1 升级到 Ionic 2

如果你有一个 App 是 Ionic 1 的，你可能想将它升级到 Ionic 2。不幸的是，这个过程并不简单。无论 Ionic 框架还是 Angular 都发生了太多变化，这使得迁移变得不太切合实际。我们的几个 App 也面临着同样的问题。根据我们的经验，这里给出了一些 App 重构的步骤。

## 创建一个新的 Ionic 2 App

要升级你的 Ionic 1App，最简单的方法是从头开始。因为改进和变化实在太多，重新创建反而更容易一点。选择最适合你的 App 的导航结构模板，用 CLI 生成一个新的 App。

### 创建页面结构

创建好 App 的框架，我们建议的第二个步骤是创建页面结构。这个步骤就好比建造房屋的框架，而前一个步骤则是浇筑房屋的地基。首先，删除由 CLI 生成的所有页面，创建你的 App 的临时占位页面。使用 Ionic CLI 的 `$ ionic g page` 命令能让你轻松许多。如果你创建的是一个 tab-base App，牢记 generate 命令会很有用。

### 修改样式

Ionic 1 也可以使用 SCSS，如果你启用了它的话。现在来看一下在 Ionic 2 中

CSS 是如何生成的。一个重要的 *.scss* 文件位于 *src/theme/variables.scss*，这里修改的样式将被应用到全局。另一个 Ionic 2 中的改变是，每个页面现在都有自己的 *.scss* 文件，你可以用它来由针对性地修改某些样式。Ionic 2 还暴露了许多在 Theming 框架中的变量，有一些在 Ionic 1 中不得不存在的东西现在不再需要了。

## 替换控制器和视图

你可能会遇到的一个最大的难题是重写所有的控制器和视图。Angular 1 的 $scope 概念被新的类和组件模型所替代。在继续转换你的 App 之前，你需要花时间理解这种新的结构，每个人都认为这种方式更加清晰和合理。如果你拥有传统的基于类的编程经验，你会发现似曾相识。此外 TypeScript 会让你提高代码的质量。

我们的视图（即 HTML）也需要做一点点改动。一些组件和 Ionic 1 中保持一致，也有另外一些，比如 cards 和 buttons 会有些许改变。另外一个改变是，每个页面中都需要包含 `<ion-header>` 和 `<ion-navbar>`。在 Ionic 1 中，这些是通过全局设置的，事实证明它要满足不同的设计要求是很难的。

## 替换你的 Factoriy 或 Service

在 Ionic 1，共享数据和功能是通过 factory 或 service 进行的。现在，这是通过在标准的 ES6/TypeScript 类前面添加一个 `@Injectable` 修饰符进行的。通过这个修饰符，Angular 会允许该类以依赖注入的方式被使用。

## 修改 App 的初始化

在 Ionic 2 中，App 的初始化方法也发生了改变。不再有用 root 来执行代码的概念了，而代之以主组件（master component）的概念。事实上，在 Ionic 2 中，整个 App 都是由组件构成的。现在，我们需要用入口组件来进行初始化。

## 修改路由

在 Ionic 1，页面路由系统是基于 UI-router 模块的。它是对默认路由的一个改进，要让它像我们在移动 App 看到的同样方式进行导航是困难的。在 Ionic 2，导航

系统学习了原生的 push/pop 风格。那些 Ionic 1 App 中存在的长长的代码块可以忽略了。现在你的导航系统能够分别处理一个屏幕一个屏幕的导航，而不是全局统一的方式。

如果你仍然需要在你的 App 中使用可以导航的 URL，请寻求 DeepLinking 系统的帮助。这是通过在 app.module.ts 文件中进行简单配置来实现的。

## 切换到 Ionic Native

如果你要在 App 中使用任何 Cordova 插件，你可能会在代码中使用 ngCordova 模块，并通过 Angular 的方式来调用它们。Ionic 团队用 Ionic Native 替换掉了该模块。这个新的代码库现在是基于 TypeScript 的，支持 Promise 和 Observable。它的文档会指导你完成这个切换。

## 结束

这只是一个将 App 从 Ionic 1 迁移到 Ioinc 2 中的核心步骤的最简单介绍。这是一个挑战，但代价是值得的。我们建议在升级你的 App 之前，先阅读完整本书并熟悉 Ionic 2 的工作机制。

附录 B

# 理解 config.xml 文件

Cordova 使用 *config.xml* 文件控制它的编译设置。当你搭建 Ionic App 的脚手架时，会为你生成一个基本的 *config.xml* 文件。*config.xml* 文件符合 W3C 的 widget 规范（*https://www.w3.org/TR/widgets/*）。它允许开发者简单地配置 App 的元数据。这篇附录解释了该文件中的各种元素，以及如何针对你的 App 配置它们。

## 基本属性

我们的 *config.xml* 的根元素是 widget 元素。它支持下面几个属性：

*ID*

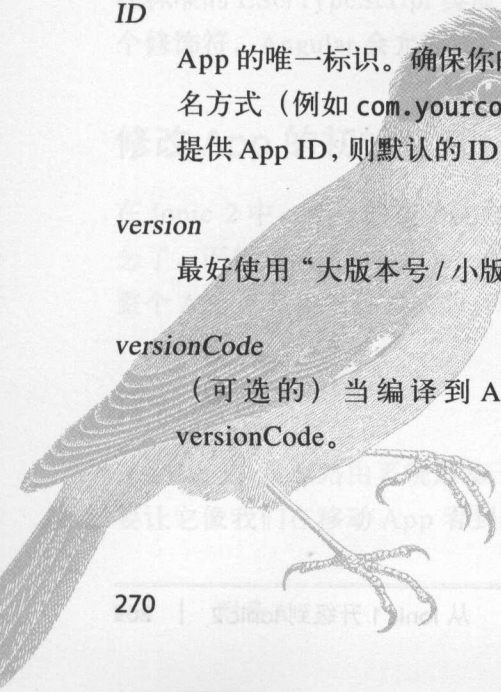
App 的唯一标识。确保你的 ID 适用于各种所支持的平台，必须是反域名命名方式（例如 `com.yourcompany.yourapp`）。如果你不使用 Ionic CLI 命令提供 App ID，则默认的 ID 会是：`com.ionicframework.[app name]`+随机数。

*version*

最好使用“大版本号 / 小版本号 / 补丁号”三个数值这样的格式，比如 0.0.1。

*versionCode*

（可选的）当编译到 Android 平台时，你可以在 config.xml 中设置 versionCode。



*CFBundleVersion*

（可选的）当编译到 iOS 平台时，你可以针对 iOS 设置这个版本。

*packageVersion*

（可选的）当编译到 Windows 平台时，可以针对 Windows 设置这个版本。

*packageName*

（可选的）当编译到 Windows 平台时，可以定义包名。

```
<widget id="com.ionicframework.ionic2do146695" version="0.0.1"↵
xmlns="http://www.w3.org/ns/widgets" ↵
xmlns:cdv="http://cordova.apache.org/ns/1.0">
```

在 widget 节点中，可以定义如下节点：

*<name>*

App 的显示名称：

```
<name>Ionic2Do</name>
```

*<description>*

App 的一般性描述。这是规范的一部分，但不会被 App 商店所引用：

```
<description>Standard To Do app using Ionic 2.</description>
```

*<author>*

App 的创建者，公司或个人（对于编译 Windows 10 平台来说，是必要的）。这个节点有两个属性需要设置：

*email*

创建者的邮箱。

*href*

公司主页或者 App 主页：

```
<author email="chrisgriffith@gmail.com" href="http://ajsoftware.com/">↵
Chris Griffith</author>
```

*<content>*

这个节点用于指定 Cordova 要加载的第一个 html 页面。不建议将 `src` 属性修改为 *index.html* 以外的内容。

```
<content src="index.html"/>
```

*<access>、<allow-navigation>和<allow-intent>*

这些节点用于定义App允许访问的多个外部域名。强烈建议你阅读并理解白名单指南（*http://bit.ly/2lcrhmx*）中的各种设置。

```
<access origin="*"/>
<allow-intent href="http://*/*"/>
<allow-navigation href="http://example.com/*" />
<allow-intent href="https://*/*"/>
<allow-intent href="tel:*"/>
<allow-intent href="sms:*"/>
<allow-intent href="mailto:*"/>
<allow-intent href="geo:*"/>
```

*<platform>*

通过将name属性设置为`ios`、`android`或`winphone`，你可以管理平台相关设置，比如权限、图标和启动图片：

```
<platform name="ios" />
<platform name="android" />
<platform name="winphone" />
```

# Preferences

Cordova通过`<preference>`标签来定义应用程序设置。这些设置可以以全局方式应用到所有平台，也有可能只针对特定平台。这些选项通过键值对属性的方式设置。

## 通用设置

```
<preference name="DisallowOverscroll" value="true"/>
<preference name="Fullscreen" value="true" />
<preference name="BackgroundColor" value="0xff0000ff"/>
<preference name="Orientation" value="portrait" />
```

*DisallowOverscroll*

如果你不想让用户向上滚动到content头部以上或向下滚动到content底部以下的时候显示动画效果的话，将这个属性设置为`true`。在iOS上，overscroll手势会导致content弹回原来的位置。在Android，这会在content顶部或底部显示一个轻微的光晕效果。

*Fullscreen*

允许你隐藏屏幕顶端的状态栏。

*BackgroundColor*

设置 App 的背景色。支持 4 字节十六进制颜色值，第一字节表示 alpha 通道，后三个字节表示 RGB 颜色值（只对 Android 和 Windows 有效）。

*Orientation*

可能取值：default、landscape 和 portrait。

这个属性允许你锁定屏幕方向，防止 UI 随屏幕方向的改变而旋转。

## iOS 通用设置

```
<preference name="BackupWebStorage" value="none"/>
<preference name="target-device" value="universal" />
<preference name="deployment-target" value="7.0" />
<preference name="SuppressesLongPressGesture" value="true" />
<preference name="Suppresses3DTouchGesture" value="true" />
```

*BackupWebStorage*

可能取值：none、local 和 cloud。

设置为 cloud 时，允许通过 iCloud 备份 Web 存储数据。设置为 local 时，允许通过 iTunes sync 进行本地备份。设置为 none 时，禁止 Web 存储备份。

*target-device*

可能取值：handset、tablet 和 universal。

这个属性和 Xcode 项目中的 TARGETED*DEVICE*FAMILY 对应。注意，如果将它设置为 universal（默认值），你必须同时提供 iPhone 和 iPad 的屏幕截图，否则你的 App 会被拒绝。

*deployment-target*

这个属性用于设置编译中的 IPHONEOS*DEPLOYMENT*TARGET，相当于 *ipa* 中的 MinimumOSVersion。

*SuppressesLongPressGesture*

将这个属性设置为 `true`，可以禁止 iOS 9+ 在用户长按 Web view 时显示一个放大镜。

*Suppresses3DTouchGesture*

将这个属性设置为 `true`，可以禁止用户在支持 3D Touch 特性的 iOS 设备上长按 Web view 并按压屏幕时显示一个放大镜图标。

## Android 通用设置

```
<preference name="android-minSdkVersion" value="16"/>
<preference name="android-maxSdkVersion" value="22"/>
<preference name="android-targetSdkVersion" value="20"/>
```

*android-minSdkVersion*

设置 Android 项目中 *Android-Manifest.xml* 文件中的 `<uses-sdk>` 的 `minSdkVersion` 属性。

*android-maxSdkVersion*

设置 Android 项目中 *Android-Manifest.xml* 文件中的 `<uses-sdk>` 的 `maxSdkVersion` 属性。不建议使用这个属性，除非你明确知道某个 Android 版本中确实会有问题。

*android-targetSdkVersion*

设置 Android 项目中 *Android-Manifest.xml* 文件中的 `<uses-sdk>` 的 `targetSdkVersion` 属性。

## Windows 通用设置

```
<preference name="windows-phone-target-version" value="8.1" />
<preference name="windows-target-version" value="8.1" />
<preference name="WindowsStoreIdentityName"
value="Cordova.Example.ApplicationDataSample" />
<preference name="WindowsStorePublisherName" value="AJ Software" />
```

*windows-phone-target-version*

设置这个 App 的目标 Windows Phone 版本。如果设置为 none，那么将使用 windows-target-version 所提供的版本（如果有的话）。

*windows-target-version*

设置这个 App 的目标 Windows Phone 版本。如果设置为 none，默认将使用 `8.1`。

*WindowsStoreIdentityName*

在 Windows 商店中的唯一标识名。

*WindowsStorePublisherName*

出版商显示名称是在 Windows 商店中列在你的 App 下方的名称。

这些只是你可以控制的众多偏好设置中的一部分。完整列表需要参考 Cordova 网站（*http://bit.ly/2lcIADN*）。

## Icons

`<icon>` 元素用于定义 App 图标。每个平台都需要不同大小的多个图标。通常每个平台的图标集合被放在一个 `<platform>` 标签中。需要注意的是，`src` 属性的值是相对于项目根目录的，而不是相对于 *www* 目录的。

## Android

```
<platform name="android">
    <!--
        ldpi    : 36x36 px
        mdpi    : 48x48 px
        hdpi    : 72x72 px
        xhdpi   : 96x96 px
        xxhdpi  : 144x144 px
        xxxhdpi : 192x192 px
    -->
    <icon src="res/android/ldpi.png" qualifier="ldpi" />
        <icon src="res/android/mdpi.png" qualifier="mdpi" />
        <icon src="res/android/hdpi.png" qualifier="hdpi" />
        <icon src="res/android/xhdpi.png" qualifier="xhdpi" />
        <icon src="res/android/xxhdpi.png" qualifier="xxhdpi" />
        <icon src="res/android/xxxhdpi.png" qualifier="xxxhdpi" />
    </platform>
```

## iOS

```
<platform name="ios">
  <!-- iOS 8.0+ -->
```

```
    <!-- iPhone 6 Plus  -->
    <icon src="res/ios/icon-60@3x.png" width="180" height="180" />
    <!-- iOS 7.0+ -->
    <!-- iPhone / iPod Touch  -->
    <icon src="res/ios/icon-60.png" width="60" height="60" />
    <icon src="res/ios/icon-60@2x.png" width="120" height="120" />
    <!-- iPad -->
    <icon src="res/ios/icon-76.png" width="76" height="76" />
    <icon src="res/ios/icon-76@2x.png" width="152" height="152" />
    <!-- iOS 6.1 -->
    <!-- Spotlight Icon -->
    <icon src="res/ios/icon-40.png" width="40" height="40" />
    <icon src="res/ios/icon-40@2x.png" width="80" height="80" />
    <!-- iPhone / iPod Touch -->
    <icon src="res/ios/icon.png" width="57" height="57" />
    <icon src="res/ios/icon@2x.png" width="114" height="114" />
    <!-- iPad -->
    <icon src="res/ios/icon-72.png" width="72" height="72" />
    <icon src="res/ios/icon-72@2x.png" width="144" height="144" />
    <!-- iPhone Spotlight and Settings Icon -->
    <icon src="res/ios/icon-small.png" width="29" height="29" />
    <icon src="res/ios/icon-small@2x.png" width="58" height="58" />
    <!-- iPad Spotlight and Settings Icon -->
    <icon src="res/ios/icon-50.png" width="50" height="50" />
    <icon src="res/ios/icon-50@2x.png" width="100" height="100" />
  </platform>
```

## Windows

```
  <platform name="windows">
    <icon src="res/windows/storelogo.png" target="StoreLogo" />
    <icon src="res/windows/smalllogo.png" target="Square30x30Logo" />
    <icon src="res/Windows/Square44x44Logo.png" target="Square44x44Logo" />
    <icon src="res/Windows/Square70x70Logo.png" target="Square70x70Logo" />
    <icon src="res/Windows/Square71x71Logo.png" target="Square71x71Logo" />
    <icon src="res/Windows/Square150x150Logo.png" target="Square150x150Logo" />
    <icon src="res/Windows/Square310x310Logo.png" target="Square310x310Logo" />
    <icon src="res/Windows/Wide310x150Logo.png" target="Wide310x150Logo" />
  </platform>
```

# Splashscreens

在 App 启动时，它会显示一个初始的启动画面，给用户一个快速的响应，然后 App 继续启动过程。和 `<icon>` 元素一样，这也是被放在一个 `<platform>` 标签中的。需要注意的是，`src` 属性是相对于项目根目录的，而不是 *www* 目录。

## Android

```
<platform name="android">
  <splash src="res/screen/android/splash-land-hdpi.png" qualifier="land-hdpi"/>
  <splash src="res/screen/android/splash-land-ldpi.png" qualifier="land-ldpi"/>
  <splash src="res/screen/android/splash-land-mdpi.png" qualifier="land-mdpi"/>
  <splash src="res/screen/android/splash-land-xhdpi.png"
  qualifier="land-xhdpi"/>
  <splash src="res/screen/android/splash-land-xxhdpi.png"
  qualifier="land-xxhdpi"/>
  <splash src="res/screen/android/splash-land-xxxhdpi.png"
  qualifier="land-xxxhdpi"/>
  <splash src="res/screen/android/splash-port-hdpi.png" qualifier="port-hdpi"/>
  <splash src="res/screen/android/splash-port-ldpi.png" qualifier="port-ldpi"/>
  <splash src="res/screen/android/splash-port-mdpi.png" qualifier="port-mdpi"/>
  <splash src="res/screen/android/splash-port-xhdpi.png" qualifier="port-xhdpi"/>
  <splash src="res/screen/android/splash-port-xxhdpi.png"
  qualifier="port-xxhdpi"/>
  <splash src="res/screen/android/splash-port-xxxhdpi.png"
  qualifier="port-xxxhdpi"/>
</platform>
```

对于你的 App 所不支持的方向，不需要提供启动图片。

## iOS

```
<platform name="ios">
  <splash src="res/screen/ios/Default~iphone.png" width="320" height="480"/>
  <splash src="res/screen/ios/Default@2x~iphone.png" width="640" height="960"/>
  <splash src="res/screen/ios/Default-Portrait~ipad.png"
  width="768" height="1024"/>
  <splash src="res/screen/ios/Default-Portrait@2x~ipad.png"
  width="1536" height="2048"/>
  <splash src="res/screen/ios/Default-Landscape~ipad.png"
  width="1024" height="768"/>
  <splash src="res/screen/ios/Default-Landscape@2x~ipad.png"
  width="2048" height="1536"/>
  <splash src="res/screen/ios/Default-568h@2x~iphone.png"
  width="640" height="1136"/>
  <splash src="res/screen/ios/Default-667h.png" width="750" height="1334"/>
  <splash src="res/screen/ios/Default-736h.png" width="1242" height="2208"/>
  <splash src="res/screen/ios/Default-Landscape-736h.png"
  width="2208" height="1242"/>
</platform>
```

## Windows

```
<platform name="windows">
  <splash src="res/screen/windows/splashscreen.png" width="620" height="300"/>
  <splash src="res/screen/windows/splashscreenphone.png"
  width="1152" height="1920"/>
</platform>
```

## Plugins

Cordova 通过插件来扩展自己的功能。通过 Ionic CLI 能够管理 Cordova 插件的安装，它会自动修改 *config.xml* 文件。默认，下列插件会自动安装到你的项目中：

```
<plugin name="cordova-plugin-device" spec="~1.1.2"/>
<plugin name="cordova-plugin-console" spec="~1.0.3"/>
<plugin name="cordova-plugin-whitelist" spec="~1.2.2"/>
<plugin name="cordova-plugin-splashscreen" spec="~3.2.2"/>
<plugin name="cordova-plugin-statusbar" spec="~2.1.3"/>
<plugin name="ionic-plugin-keyboard" spec="~2.2.1"/>
```

*name*

插件是通过插件 ID 来引用的，通常它是以反域名格式命名的（例如，`com.phonegap.plugins.barcodescanner`）。

*spec*

可选的，但我们强烈建议用它来指定插件的版本。

有的插件可能需要额外的参数。具体请查看每个插件的说明文档。

## Features

这个元素专用于那些需要在 Web view 初始化过程用进行初始化的平台相关插件：

```
<feature name="StatusBar">
  <param name="ios-package" onload="true" value="CDVStatusBar"/>
</feature>
```

*name*

可能取值：android-package、ios-package 和 osx-package。

*onload*

用于表示对应插件（通过 value 属性指定）是否需要在 controller 初始化时进行初始化。

*value*

包名，用于初始化这个插件的代码。

## 参考

*config.xml* 文件的完整说明请参考 Cordova 网站（*http://bit.ly/2l8zSqd*）。

# 附录 C

# Ionic 组件库

Ionic App 基于一系列组件构建。这些构建材料本质上是 HTML 和 CSS，然后用一些 JavaScrpit 使它们具备相应的功能。本附录简单地描述每个组件，让你对它们有一个基本的认识。

*Action Sheets*

这个组件以从屏幕底部向上弹起的方式显示一系列操作。这个组件直接用代码创建，不需要任何 HTML。

*Alerts*

尽管可以通过 Dialogs 插件来使用原生对话框，但当你想显示更复杂的 alert（比如要在上面显示一颗选项按钮或者检查框），或者 Dialogs 插件不可用的时候，你可以使用这个组件。这个组件可以模拟所有平台的原生对话框。这个组件直接用代码创建，不需要任何 HTML。

*Badges*

这个组件用于标注某些东西的数目（比如未读通知数）。它的每个颜色属性都可以设置。

*Buttons*

这是 App 中用得最多的组件之一。它支持各种各样的样式：default、Outline、Clear、Round、Block、Full、Icon 和 Floating。通过它的标准属性，可以轻易修改它的颜色和大小。

除了单独使用这个组件，也可以在其他 Ionic 组件比如 Toolbar 和 Card 中使用它。

*Cards*

Card 组件是一种常见的用于组织和包纳信息的 UI 组件。这个组件可以做非常灵活的设计，包括头部、占据全宽的图片、按钮行等。

*Checkbox*

这种标准的 input 类型用于保存布尔值。这是一个 HTML 组件的 Ionic 版本，会自动根据平台来适配样式。

*DateTime*

这是 Ionic Framework 中新加的一个组件。它允许用户轻松地输入日期和时间。日期和时间的格式是可配置的。

*Gestures*

Ionic 框架中内置的手势系统。它允许每个组件能够响应一系列标准手势：轻触、点击、平移、轻扫、旋转和捏合。

*Grid*

Ionic 的网格系统使用 CSS 的弹性盒子模型构建，它包含了三个元素：grid、row 和 column。通过弹性盒子布局系统，grid 的内容很容易实现垂直对齐效果。

*Icons*

Ionic 框架内置了 700 多个图标（而且还会增加）。这个组件支持 active 和 inactive 两种状态，并自动适应目标平台。

*Inputs*

它是标准 HTML 标签的 Ionic 版本，自动根据对应移动平台进行样式和功能的适配。它支持各种风格：Fixed Inline 标签、Floating 标签、Inline 标签、Inset 标签、Placeholder 标签和 Stacked 标签。

*Lists*

List 组件可能是你使用得最多的组件之一，仅次于 button。这个组件适合

用于显示滚动列表。除了基本列表样式外，还可以指定其他样式：Inset、No Lines、Multi Line 以及 Sliding。

列表头和分割线是可配置的。和其他 Ionic 控件比如 Toggle 一样，List 也很容易呈现图标和头像。

*Loading*

这个组件用于显示加载状态。旋转动画支持一系列风格。这个组件通过代码创建，不需要任何 HTML。

*Menus*

这个组件用于创建一个侧滑式菜单。样式和呈现方式根据目标平台自动适应。这个组件因为需要和导航及 App 中的其他元素相关，因此显得比较复杂。如果你打算在 App 中使用它，请花点时间读一下 API 文档。

*Modals*

Modal 组件是另一种实现对话框的方式。它会滑入屏幕并显示它的内容。这个组件通过代码来创建，不需要使用任何 HTML。

*Navigation*

Ionic App 可以用 <ion-nav> 组件来处理在 App 中的一系列导航。它使用一种标准的 push/pop 模型。返回按钮和屏幕标题会根据 App 的“流动”而改变。

*Popover*

Popover 组件会显示在 App 的内容之上。这个组件常常被用来进行某种设置或过滤器的切换。这个组件是通过代码而不是 HTML 来创建的。

*Radio*

和检查框类似，Radio 组件也用来保存布尔值。Ionic 的 Radio 组件和 HTML 的 radio 控件并无不同。但是，和其他 Ionic 组件一样，它的样式会针对不同平台进行适配。

*Range*

也叫做 Slider 组件，这个 Ionic 组件允许用户在一个固定范围中进行选择。

*Searchbar*

这个组件是和平台相关的，允许用户输入一个搜索用的关键字。通常，这个组件和 list 组件配合使用。

*Segment*

这个组件也叫做 button-bar 组件，在一行中显示多个按钮。

*Select*

`<ion-select>` 组件类似于标准的 HTML 标签 `<select>`。和其他增强控件一样，它自适应 App 所运行的平台。

*Slides*

Slide 组件是一个对移动设备友好的图片轮播器。每个 slide 都在 <ion-slides> 标签中指定。它提供一系列内置功能，比如自动播放、方向、循环及分页。

*Split Pane*

这个布局类组件允许你定义两栏式的界面，一个菜单式的面板和一个主内容面板。由于 viewport 的宽度变窄，菜单栏会收缩，就像一个侧滑菜单。这个组件在平板或桌面应用的界面中比较常用。

*Tabs*

这个布局类组件允许定义多个 tab，每个 tab 都拥有自己的导航栈。tab 可以显示文字或图标之一，或者二者都显示。它们的外观和行为根据所运行的目标平台而定。这个组件混合了 HTML 和 TypeScript。

*Toast*

这个组件通常用于在 App 显示内容之上显示简单信息。这个组件通过代码而不是 HTML 来创建。

*Toggle*

Toggle 是一个简单两状态的开关。当需要提供一个用于开 / 关某种特性的 UI 元素时，最好使用这个组件。

*Toolbar*

这个组件就是一个普通的 bar，有几种使用方法：header、sub-header 或者 footer。这形成了三个版本：`<ion-header>`、`<ion-footer>` 和 `<ion-toolbar>`。当需要一个 header 或者 footer 时，建议使用 `<ion-header>` 和 `<ion-footer>`。这个组件中可以放入图标、按钮、segment 和 searchbar。

## 作者介绍

**Chris Griffith** 是一个家庭自动化和安全公司的 UE 负责人，同时也是加利福尼亚大学圣地亚哥分校的讲师，讲授移动 App 开发。他还是一名 Adobe 社区的专家，尤其专注于 PhoneGap/Cordova 和体验设计。Chris 经常受邀在 Fluent、Adobe Max 和 ngConf 会议上做发言。他开发了多个移动应用，为 PhoneGap Build 开发了各种代码提示器以及 ConfiGAP。此外，他还是 uxmag.com 的作者和多个出版物的技术评稿人。在业余时间，Charis 喜欢和家人一起，划海上皮艇、徒步旅行，以及和朋友喝精酿啤酒。你可以关注他的 Twitter（*@chrisgri th*）或者 *chrisgriffith.wordpress.com* 网站。

## 封面介绍

本书封面上的动物是一只欧亚黑顶或黑顶林莺（学名 Sylvia atricapilla）。

黑顶林莺因头部冠羽的颜色而得名，雄性为黑色，雌性为红褐色。躯干颜色多为灰色，上部为灰橄榄色，下部为浅灰色。雄鸟鸣声圆润动听，有柔和的颤音，结尾时以响亮的、尖锐的渐强音结束。在许多孤立栖息地比如阿尔卑斯能够听到它的简短的鸣叫。由于它优美的鸣叫声，黑背林莺也被称作“夜莺的模仿者”。

黑背林莺分布在欧洲、西亚和西北非。它喜欢成熟的落叶林地，喜欢在矮荆棘或灌木丛中建造干净的巢穴。通常产卵 4~6 枚，孵化期 11 天。

它的主要威胁来自于地中海居民的捕杀以及大自然的威胁，比如捕食和疾病。尽管受到这些危害，黑背林莺还是在近些年扩大了栖息地的范围。

许多 O'Reilly 的封面动物都是濒临灭绝动物，它们是这个世界的重要一员。要进一步了解如何帮助它们，请访问 animals.oreilly.com 网站。

封面图片来自于英国鸟类月刊。

O'REILLY®

# 基于Ionic的移动App开发

学习如何用Ionic 2框架来编写用于提交到应用商店的混合App，该框架基于Apache Cordova（前身为PhoneGap）和Angular构建。这是一本很特别的教程，教你如何用Ionic的工具和服务开发用HTML、CSS和TypeScript编写的App，而不是针对特定平台如Android、iOS和Windows Phone的App。

本书作者以step by step的方式带你感受Ionic强大的UI组件库，教你用它编写三个跨平台移动App。无论你是一个Ionic新手，还是曾经使用过Ionic 1，这本书对初、中、高级Web开发者来说都是不错的选择。

- 理解什么是混合移动App，以及Ionic App的基本构成。
- 学习如何用Apache Cordova、Angular和TypeScript创建一个原生移动App。
- 创建一个基于Firebase的to-do App，存储跨过多个客户端的数据。
- 编写一个基于tab的国家公园App，集成Google地图。
- 开发一个天气App，调用Darksky的天气API和Google的地理编码API。
- 调试和测试App，解决开发中出现的问题。
- 了解将App发布到原生应用商店的步骤。
- 学习如何用Ionic创建渐进式Web App。

"Ionic 2是一个伟大的、稍有点复杂的框架，它从Ionic 1升级而来。本书作者极好地帮助你了解到这些改变，学习使用Ionic的最新版本。"

——**Raymod Camden**
IBM的Developer Advocate，
Cat Demos的作者

**Chris Griffith**是一家家庭自动化和安全公司的UE负责人，同时也是加利福尼亚大学圣地亚哥分校的讲师，讲授移动App开发。他也是一个Adobe社区的专家，尤其专注于PhoneGap/Cordova和体验设计。Chris经常受邀在 Fluent、Adobe Max和ngConf会议上做发言。

PROGRAMMING / APP DEVELOPMENT

ISBN 978-7-5198-1424-3

O'Reilly Media, Inc.授权中国电力出版社出版



定价：68.00元